FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

API design and implementation of a management interface for SDN whitebox switches

Rubens Jesus Alves Figueiredo

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Dr. Ana Cristina Costa Aguiar

External Supervisor: Dr. Hagen Woesner

April 14, 2018

Resumo

Os requisitos crescentes dos serviços em nuvem de hoje requerem a evolução da infraestrutura de rede para suportar a quantidade crescente de dados que são processados todos os dias. Isso significa que os operadores de centros de dados devem projectar ou adaptar os seus ambientes de rede em nuvem para fornecer uma conexão estável e confiável. Uma infraestrutura otimizada muitas vezes significa também a redução de custos na utilização dos sistemas e nos gastos em energia.

À medida que as redes crescem e são mais complexas, os sistemas devem ser implementados de forma a permitir não só acompanhar de perto os recursos que a compõem mas também permitindo uma certa liberdade para a possível evolução dos requisitos. Como tal, as soluções típicas dos fornecedores não se encaixam realmente nessa paisagem de constante mudança, uma vez que apresentam soluções muito sólidas e verticalmente integradas. O paradigma do Software Defined Networking, no entanto, é capaz de resolver esse problema, pois permite o controlo centralizado das redes subjacentes, proporcionando visibilidade e controlo sobre os dispositivos, simplificando o diagnóstico de erros e proporcionando uma maior capacidade de resposta.

Neste trabalho, propomos um sistema de gerenciamento modular para controladores de Software Defined Networks dos centros de dados da nuvem, fornecendo aos administradores de sistemas uma plataforma simples, destacando-se e ressaltando a visualização da topologia de rede e monitorização das portas dos dispositivos. A modularidade também fornece uma plataforma simples para estender a funcionalidade dos controladores de rede, que podem ser usados para implementar a detecção de anormalidades e optimizar os caminhos de encaminhamento de fluxo, entre outros.

Abstract

The rising requirements of today's cloud services require the evolution of networking infrastructure to support the increasing amount of data that is processed every day. This means that data center network operators must design or adapt their cloud networking environments to provide a stable and reliable connection. Better optimized infrastructure often also means cost reductions in network utilization and energy savings.

As networks grow larger and more complex, systems must be put in place that allow for closely monitoring the resources that make up the network, while also allowing for a certain freedom for the possible constant change of the network. As such, typical vendor solutions don't really fit into this ever changing landscape, since they present very solid and vertically integrated solutions. The Software Defined Networking paradigm, however, is able to solve this issue, since it enables the centralized control of the underlying networks, providing visibility and control over the network's devices, simplifying error diagnosis and troubleshooting.

In this work we propose a modular management system for cloud data center Software Defined Networking controllers, providing system administrators a simple platform to view their network's topology, monitor networking devices ports, etc. The modularity also provides a simple platform to extend the functionality of the networking controllers, that can be used to implement detection of network abnormalities and optimize flow forwarding paths, among others.

Acknowledgments

First of all, I'd like to thank my supervisor Dr. Ana Aguiar, for the constant support and guidance that was provided. I'd would also like to thank for the opportunity of finishing my master's thesis in a foreign company, and spending these past few months in Germany was an amazing experience.

The constant environment of teaching and support provided by Dr. Hagen Woesner and the team at BISDN was undoubtedly a big factor to my adjustment in Germany. The experience I gained there, both technically and personally was a big mark during the development of this thesis, and the working atmosphere contributed immensely for my integration there.

Next I would like to thank my mom, my dad and my brother, for the immense support that they showed during my time here and abroad, and the understanding that only they could provide.

Finally, I thank my group of friends that helped me grow during these past few years, and the motivation they always gave me.

Rubens Figueiredo

"By the time you've sorted out a complicated idea
into little steps that even a stupid machine can deal with,
youve certainly learned something about it yourself.

Douglas Adams

Contents

List of Figures

List of Tables

Abbreviations & Acronyms

API	Application Programming Interfaces
ASIC	Application-specific integrated circuits
CAWR	Capability Aware Routing
FPGA	Field-Programmable gate arrays
GUI	Graphical User Interface
IETF	Internet Engineering Task Force
LACP	Link Aggregation Control Protocol
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
NFV	Network Function Virtualisation
ODL	OpenDaylight
OF-DPA	OpenFlow Data Path Abstraction
RPC	Remote Procedure Call
SAL	Service Abstraction Layer
SDN	Software Defined Networks
TE	Traffic Engineering
VLAN	Virtual Local Area Network

Introduction

1.1 Context

The rapid expansion of the cloud computing environment in the previous decade is related to the increasing demand in computational power that applications like distributed databases or data analysis have. Public cloud solutions like Amazon's Web Services, Microsoft's Azure, or private solutions offered through OpenStack provide a very large pool of resources for developers to deploy applications with ease. Through economies of scale, these vendors have centralized their solutions in very large scale data centers, consolidating their operations in a single location, allowing increased performance of applications, and easier maintenance. The offer of virtualisation solutions also contributes to the recent surge in popularity of these systems, due to the less spending in operational costs and improved utilization of hardware [1]. Subscription based systems are typically available for renting, allowing users to use services on a Virtual Machine.

Using open source applications and whitebox hardware has also contributed to the success of these environments, due to the possible cost reductions. Software Defined Networking (SDN), has proven to be a reliable environment to manage data center environments, due to the centralization of the network controllers, improved programmability of the network's data plane, and improved management systems. Network programmability, despite not being exclusive to the SDN framework, eliminates the effort in individually configuring every network device, which in large scale environments becomes an impossible task. This model also provides the network engineers and software developers a high level network abstraction that is used to monitor network utilization and optimize resource utilization.

Network management systems (NMSs) provide the central architectural component that allows the system administrators to track the systems utilization, analyse link, node and device failures, and observe alarms when the networks' state is outside the range that network operators define as normal. These systems also provide an insight for network operators to research where resources should be allocated. However, network planning considering virtualisation deployments is considered a difficult task, due to complex scheduling policies and performance deviations [2].

The topic of traffic engineering in SDN is central to this field, more specifically, the flow and application level monitoring. The possibility of acting differently towards network traffic is an advantage

for network operators, allowing to prioritize applications that are sensitive to latency, like video and audio streaming. As such, focus on service prioritization and delivery optimization has been the focus of research of the past few years [3].

1.2 Motivation

The increased relevance that cloud computing solutions have on today's networking environments, and the growing opportunities in data centers, mainly kick-started by the prevalence of OpenStack, have brought a rising demand of open sourced solutions that provide an interface in these environments. Exposing a programmable interface for managing networking devices is a central function of network controllers. Due to the prevalence of Linux-based environments in these data centers, network administrators' familiarity with the Linux networking stack provides a possible interface for network management.

With this consideration, BISDN developed Basebox, an open source Software-Defined Network controller that interacts between the Linux kernel's networking library and networking devices. This provides a familiar and stable Application Programming Interface to configure ports, Virtual Local Area Network (VLAN) and routes, and the advantages of using tools like systemd-networkd and iptables.

Despite providing a stable platform, Basebox also provides a

1.3 Aims and objectives

This thesis aims to plan and develop a system that exposes the physical topology of connections between switches and server, and monitoring the port configuration and statistics. This management system aims to clearly display changes in configuration and behaviour of the networking infrastructure connected to a SDN controller, which will allow system operators to maintain finer control over their resources. In this first part, we aim to define the components to implement this management system, from the necessary modifications to the existing controllers, to the Graphical User Interface that users will interact with. Typical management solutions like Icinga [1], Zabbix [2] or Graphana [3] display relevant metrics for monitoring of network devices, including the display of the physical topology, monitor the port status, display devices temperature, among others. In this stage, we define the Minimum Viable Product (MVP) for this system as a simple Graphical User Interface that can display the connections between switches and servers, and expose port statistics like the number of received packets and bytes and the number of errors.

We propose an Operations Support System (OSS) that interacts with the controllers and implements intelligence for monitoring the state of the network. Due to data center's traffic profile, one common challenge for optimizing the networks resource utilization is the asymmetry of traffic, where most of the networking flows are short-lived, latency-sensitive quick bursts of packets, but do not amount for the

[1] https://www.icinga.com/
[2] https://www.zabbix.com/
[3] https://grafana.com/

total traffic in the network. The main contributor for the total traffic volume are the large and long-lived flows, usually called *elephant flows*. Solution to this problem will increase the networks capability of splitting resources between the multiple competing virtualised applications, and, as a consequence, their associated flows. Furthermore, by maintaining a system that monitors and alarms network operators of the occurrences of large data streams, this will provide insight to the network operators to plan ahead their network resources. To implement this functionality, we researched the most common ways that detection of network traffic changes are done, and propose an algorithm that enables quick detection of changes in the network traffic behaviour via simple statistical methods.

1.4 Organization

This dissertation is divided in the following chapters:

- **Background** begins with an overview of Software-Defined Networking, providing an insight on the protocol behind these environments, and presenting information on the existing applications. We also give an overview of network management, and approach what the roles of management systems are in current data centers.
- **Related Work** explores the deeper concepts of statistical detection, providing a formal framework for change detection mechanisms. We approach the techniques traditionally used in change detection mechanisms, and evaluate the performance measures used to assess their efficiency. We also explore further the concept of elephant flows, exploring a possible mathematical representation, and present the mechanisms typically used for detection and mitigation of this network phenomenon.
- **Monitoring SDN Switches** is a chapter dedicated to defining the proposed architecture for this thesis, by defining the components and the high level approaches for implementing this solution, providing a description on the architecture of the system this thesis will integrate.
- **Management API** approaches the first stage of the developed work. First, we design the networking entities with a set of standardized data models for networks, so that the structure for creating the management interface is logically organized. Then we explore the researched alternatives for implementing the transport protocol between the network controllers and the Graphical User Interface. Finally we demonstrate the final Graphical User Interface developed for this stage.
- **Elephant Flow Monitoring** is the final chapter, where we explore the proposed solution for the OSS, more specifically the part related to the elephant flow monitoring. In this chapter we explore the design of the algorithm for monitoring the port changes, using the statistical methods presented in the Related Work chapter. Finally, we present the results of the detection algorithm, and propose a set of optimizations for use in this part.

Background

2.1 Network Management

The topic of network management is very extensive, due to the many components that make up today's networks, and the vast amount of information that they provide. It can be summed up as the operation and maintenance of network infrastructure so that the service it provides is not only "healthy", but also is operated at a level that keeps costs down for service providers.

2.1.1 Requirements for management systems

As the complexity of the networks, and network devices that compose them, grows bigger and bigger, the management systems should accommodate for the their necessities. As such, the basic groups of requirements for management functions defined in the ITU-T X 700 Recommendation [22] are:

- **Fault management** is the capability for detection, isolation and correction of abnormal operations in the system
- **Accounting management** provides ways to monitor the system resource utilization, and using this data to generate information about the costs that the operation of a certain resource will incur. This allows for optimizing the network utilization of resources, as it provides insights on how to plan the evolution of the network
- **Configuration management** is related to the maintenance and updates of hardware and software in the network, and the general setup of devices that allow to start, maintain and terminate services
- **Performance management** relates to monitor systems for the traffic utilization, response time, performance and logging histories. This allows to maintain Service Level Agreements (SLA) between the service provider and the client, providing better services even in cases of unusual traffic.
- **Security management** enables setting up security policies in terms of access control to resources, private information protection, among others.

A network management system usually consists of a centralized station, and management agents running on the network devices. Using management protocols, the agents can report to the station information about the its operational status, which includes information ranging from CPU load to bandwidth usage. Typically this information can be retrieved by the controller polling the agents, or the agents sending information on their own, usually to inform of status changes. Using this information, the network operator can get insight on the performance or possible errors of the devices that are monitored. In the next section, we explore one of the most popular management protocols, SNMP.

2.1.2 SNMP

The Simple Network Management Protocol is an IETF defined protocol that allows for the interconnection of networking devices, and provide a structured way to retrieve relevant information about these devices. As the name suggests, SNMP allows for a simplified approach to network monitoring, since it reduces the complexity of the functions that the management agent needs to comply with, which bring several advantages, like reducing the costs for development of management tools; providing a way to monitor, independently from different hardware providers the resources; and also supporting freedom in extending the protocol in order to include other aspects of network operation [23].

The architectural model of SNMP can be described in figure 2.1.

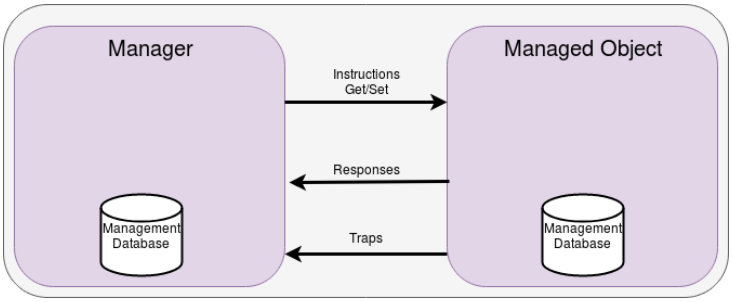


Figure 2.1: Architectural components of SNMP

The management database is one of the most important components of this system, because it serves as a reference to the entities that are managed in the SNMP protocol. The formal name for this database is the MIB - Management Information Base [24], and its composed of a collection of objects.

Each object has a name, syntax and encoding [25]. The name of the object, more specifically, the *Object Identifier (OID)*, is a reference to the object itself. This name is usually a list of integers, and they serve to build a tree-like hierarchy. This structure allows for the organization of all objects in a logical pattern, as there is a parent node that contains references to their children, which provides different indexes for different objects. For human readability, there is usually an *Object Descriptor*, to refer to the object type.

The syntax defines the type of data structure in the object type; and the encoding describes how the object type is transmitted on the network. In the context of this thesis, an important group is the interfaces

group, as it exposes information about the interfaces present in a system. Its OID is .3.6.1.2.1.2., and it contains the number of interfaces in a system, and a table containing the counters related to the interface status, like the received unicast packets, the physical address, among others. The flexibility of the MIB allows for vendors to introduce their own databases into the MIB, while also remaining compatible with the standardized one.

Due to its permanence in the market, the protocol has suffered some large changes since its original design. SNMPv3 now supports important changes to the original one, most notably in the security aspects, introducing strong authentication and encryption capabilities.

Despite it's dominance on network management products, SNMP features some bad characteristics that pose an obstacle for the widespread use in network configuration and management, like [26]:

- Incompleteness of the devices features
- SNMP access can sometimes crash systems, or return wrong data
- Unavailability of MIB modules, which forces users to use CLI's
- Poor performance
- Security is difficult to handle

2.1.3 Data Center Networks (DCN)

The design of the network architecture is central to the data-center networks, as the placement for physical hosts and virtual machines allows for sharing the resources and create a logical hierarchy of network devices. The study of the design of DCN has resulted in the creation of typical DC topologies, like fat-tree topologies (as seen in 2.2), or others, including de Bruijn server only networks, or BCube switch heavy networks [27]. Using these standardized architectures monitoring the traffic characteristics, resource consumption and costs of the networking devices are easier, and causes for failure of are understood and mitigated, and the entire DC can run on the most optimal way possible. The organization in the DCN also allows for traffic in the network being resistant to failure scenarios, since multiple paths can be used, redirecting packets to the correct destination even if a link to a switch fails.

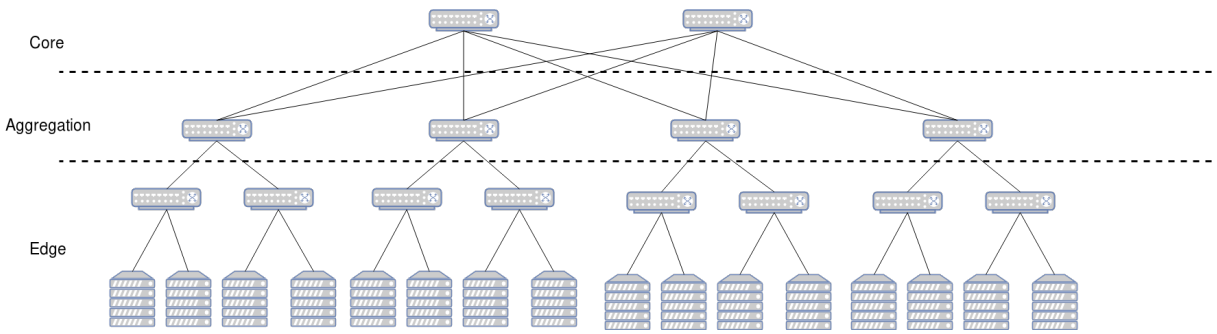


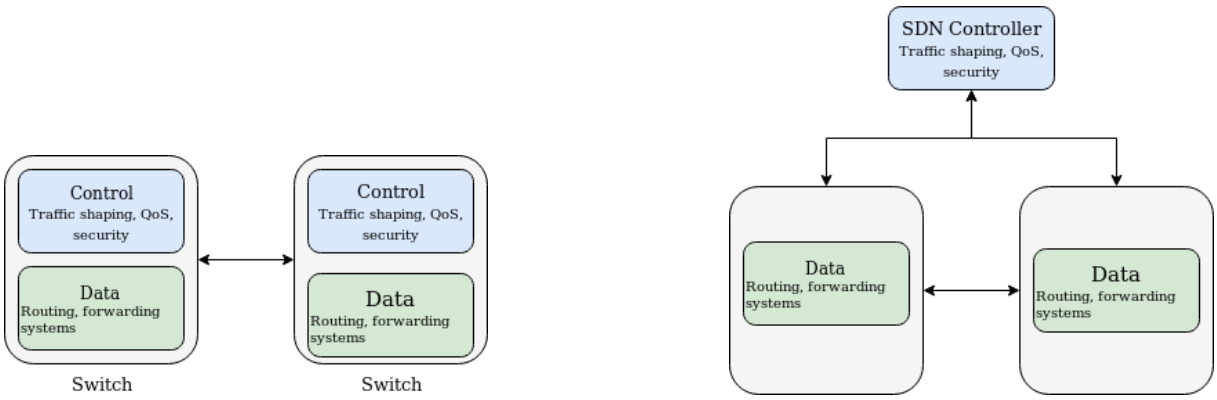
Figure 2.2: Visual representation of the fat tree topology commonly used in data centers

2.2 Software Defined Networking

Computer networking is a vital part of the services that are offered today, and, as such, the performance in technology backing these is central to the quality of these services. As the service providers move their data centers to cloud computing environments, enabling several improvements in the predictability, Quality of Service and ease of use of their services, new technologies are required to make sure that their services are adapted to the fast changing landscape of networking. One of the most notable innovations in this field is called **Software Defined Networking**, where, as described by the Open Networking Foundation, *the control and data planes are decoupled, network intelligence and state are logically centralized, and the underlying network infrastructure is abstracted from the applications* [4]. The two main contributions of this architecture are:

- **Separation of data and control planes** SDN allows for the separation of the network control plane from the data forwarding plane by having network "intelligence" present in the network controllers, and having them control the forwarding elements that live in the Data Plane
- **Centralization of network management functions** By isolating the management on a separate plane, there is possibility of developing a single controller that can regulate the entire network, having unrestricted access to every element present in the network, simplifying management, monitoring, application of QoS policies, flow optimization, etc

This new paradigm introduces programmability in the configuration and management of networks, by consolidating the control of network devices to a single central controller, achieving separation of the control and the data plane, and supporting a more dynamic and flexible infrastructure. This concept removes *middleboxes* [1] replacing them with generic software applications.



(a) Traditional networking architecture (b) SDN architecture

Figure 2.3: Traditional vs SDN network architecture

[1] Computer networking device that does some operations on traffic, besides packet forwarding. Examples include caches, Intrusion Detection System (IDS), Network Address Translation (NAT), etc

By moving network infrastructure to SDN models, the difficulty of managing a network is greatly reduced, since the logical centralization of the control layer exposes the global view of the network, simplifying management tasks. Furthermore, this also removes the challenge of configure each networking device individually, turning network operation and management into setting high level policies in the controllers, and letting the protocols that handle connection between the devices and controllers set the actual rules.

Software-Defined Networking is defined as being composed of two layers: **Northbound Interfaces**, which are composed of Application Programming Interfaces (API) for communication between applications and the controller, enabling network services like routing, security, visualization and management; and the **Southbound Interfaces** which connect the network devices to the controllers via protocols like OpenFlow (see section 2.2.1), or P4 [2].

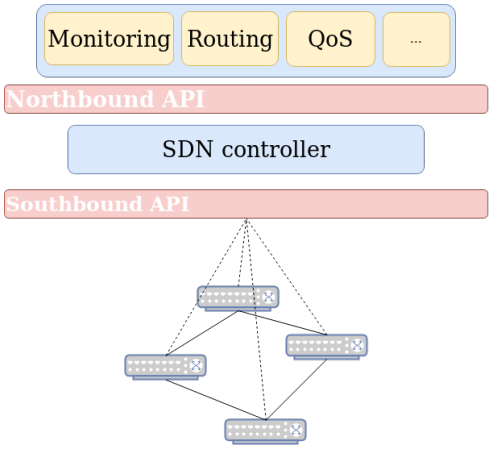


Figure 2.4: General overview of the SDN architectural components

Understanding SDN platforms is then composed of understanding the operation of both interfaces, and defining requirements for their operation, which are listed below [5]. These requirements are general principles for networks, but the addition of the SDN controller introduces a single point of failure, that could be damaging to the entire network.

1. High Performance
2. High Availability
3. Fault Tolerance
4. Monitoring
5. Programmability
6. Modularity
7. Security

[2] https://p4.org/

2.2.1 OpenFlow

With the growth of the networking infrastructure of the past few decades, the need for an environment that allows for experimentation and testing of different protocols and equipment became evident. As such, there was need for a framework that could enable testing of new ideas on close to realistic settings. So, on February 2011 OpenFlow was released, and this proposal quickly became the standard for networking in a Software Defined Network. Since 2011, this protocol has suffered some revisions, and the latest version released is version 1.5.1.

Several reasons led to the quick standardization of this protocol, which are related not only to the initial requirements of the platform, like the capability of supporting high-performance and low-cost implementations, but also the extensibility that the open source development model provides, removing the limitations that typical commercial solutions give the network researchers.

The big advantage of OpenFlow is that it is, from the data forwarding point of view, easy to process, since the control decisions are made by the controller present in a separate plane, and all the switch needs to do is correctly match the incoming packets, forwarding them according to the rules established by the controller. The components that are part of this system and enable this functionality are:

- **Flow Tables** This element describes the main component of the switching capabilities of the OpenFlow switch. Inside the switch there are several flow tables that contain rules to match incoming packets, and process them according to the rules specified by the controller. These rules can contain actions that affect the path of the packets, and these actions usually include forwarding to a port, packet modification, among others. Classification is done via matching one or more fields present in the packet, for example the switch input port, the MAC and IP addresses, IP protocol. The required actions for an OpenFlow switch are the capability of forwarding to a set of output ports, allowing the packet to move across the network; to send them to the controller, in the case of a miss of match; and finally the ability to drop packets, which is useful for DDoS mitigation, or other security concerns.

- **OpenFlow Protocol** The OpenFlow Protocol between the switch and the controller defines several messages that allow for the control of the switch. This protocol enables capabilities such as requesting the available features on the switch, configuration of flow rules, among others, using the messages referred to as *Controller-to-Switch*. Other relevant message types are the *Asynchronous* messages, that provide notifications of events that occurred. This type includes the PACKET_IN message, which is a type of message that is sent to the controller when a certain packet has no match in the flow tables present in the switch. Finally, the *Symmetric* message, like the Hello message, which are used for negotation of the OpenFlow version and other elements to help connection setup. The initial connection is initialized by either the switch or the controllers over the defined transport protocol, but after the transport connection is established, the OpenFlow channel should behave the same way [6].

- **Secure Channel** OpenFlow defines the channel that is between the switch and the controller as a secure communications channel. As messages that are exchanged between the switches and the controllers are critical for the correct operation of the system, the channel should be cryptographically secure, to prevent spoofing and manipulation of this information. As such, the channel is usually transported over TLS.

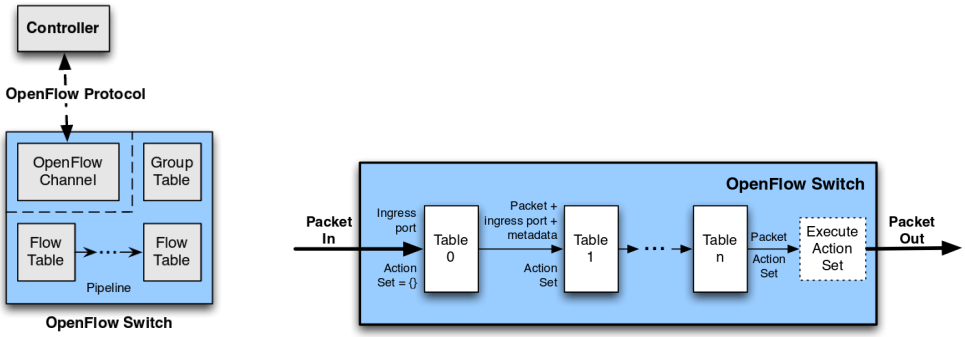


Figure 2.5: Images describing OpenFlow components. On the left, an overview to the entire system, and on the right a view at the table structure of the OpenFlow Switch [6]

Figure 2.5 describes the OpenFlow components. The image on the left shows the entire pipeline and the connections between the controllers and switch, but lacks the connection of the OpenFlow switch to the data plane switch ports. In image on the right, the structure of the Flow Table is summarized, and the packet in originates from the switches' ingress port, and after processing, the packet will exit through the switches' egress port.

In the case of controller failover, then the backup controllers should act on this failure, and act as the new master. OF switches should connect to the set of available controllers, which should coordinate the management of the switch amongst themselves, according to a set of certain roles. After the switches' first connection to the controllers, they should maintain these connections alive, but the controllers have the possibility of changing their roles. These roles are as follows:

- OFPCD_ROLE_EQUAL, where the controller has full access to the switch, receiving all incoming messages, and can modify the state of the switch
- OFPCD_ROLE_MASTER, which is a similar status to the previous one, but where the switch ensures that only one switch is connected as the master role
- OFPCD_ROLE_SLAVE is a role that controllers has read-only access to the switch, having no permissions for altering the state of the switch. The only message that controllers registered with this role receive are the port-status messages

The OpenFlow switches maintain a set of counters, similar to SNMP, that provide information about the state of the ports, group, flow and table stats. The statistics that are exposed from OF are shown in

table 2.1. In this table, the fields duration_sec and duration_nsec specify the time that each port has been configured, in seconds and nanoseconds respectively.

Table 2.1: OpenFlow port statistics

uint64_t	rx_packets	uint64_t	tx_packets;
uint64_t	rx_bytes;	uint64_t	tx_bytes;
uint64_t	rx_bytes;	uint64_t	tx_dropped;
uint64_t	rx_errors;	uint64_t	tx_errors;
uint64_t	rx_frame_err;	uint64_t	tx_over_err;
uint64_t	rx_crc_err;		
uint64_t	collisions;		
uint32_t	duration_sec;		
uint32_t	duration_nsec;		

2.2.1.1 OpenFlow Data-Path Abstraction

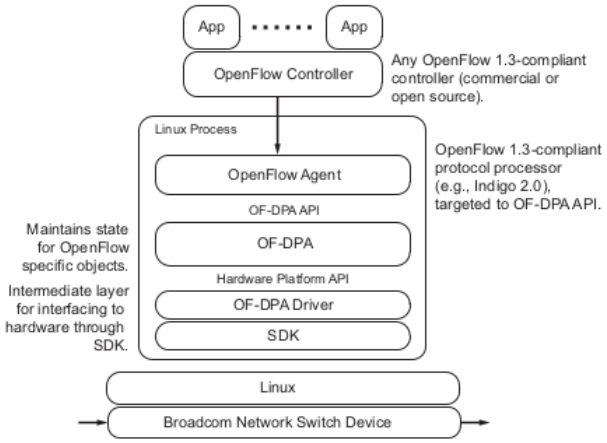


Figure 2.6: OF-DPA components [7]

OpenFlow Data-Path Abstraction (OF-DPA) allows development of OpenFlow based SDN applications based in Broadcom's hardware switches. It provides a hardware abstraction layer, supporting programming of network devices using OpenFlow. The main difference from pure OpenFlow to Broadcom's implementation are the Flow Table structures, known as the Table Type Patterns (TTP). These TTPs are templates that describe the protocol features and messages that the controllers and switches need to support, providing developers additional structure for easier implementation of their applications. Furthermore, the specialized structure provided by these patterns allow for optimizing the allocation of table memory, and improve the lookup algorithms.

Figure 2.7 displays the utilized TTP in OF-DPA.

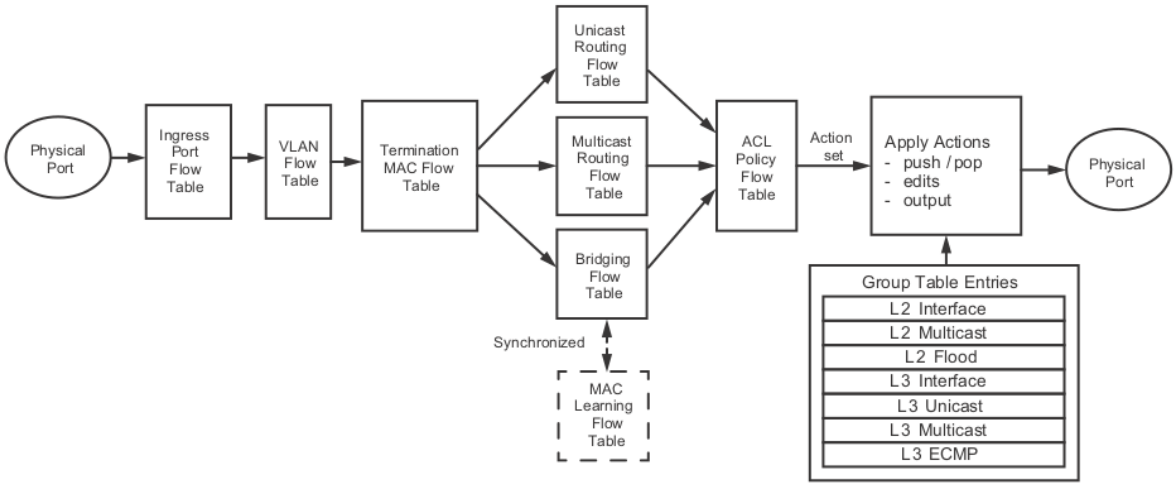


Figure 2.7: OF-DPA TTP [7]

2.2.2 Network Devices

Networking devices are a central part of network operation, performing routing, switching, management operations, and span the different layers of the OSI model. The investment in dedicated hardware to perform management functions can potentially be replaced by SDN controllers, offloading the QoS policies and traffic engineering (TE) functions from hardware to software. As such, in this section we focus on the devices that are responsible for the operation on layers 2 and 3 of the standardized networking stack, switches and routers. A networking switch is a device that connects multiple devices on a computer network, using hardware addresses (MAC) to forward data inside the network, by mapping each port with a one (or more) MAC addresses, while a router is responsible of forwarding packets between different computer networks. These devices run at different layers of the networking stack, the former operating at the data-link, or layer 2, and the latter operating at layer 3, or network. This is not a clear separation however with multilayer devices, where switches also provide routing capabilities. Typical vendors for these solutions include Cisco and Juniper, but the rise of whitebox switches that have support for deployments in SDN environments enables network operators to avoid vendor lock-in, and take advantage of the open nature of these devices. Commonly associated with whitebox switches is the support for the OpenFlow protocol, making these an essential part of the SDN infrastructures.

The performance [3] of networking devices is central to the proper operation of networks, especially in deployments in Data Centers, where the interfaces must be able to support 100 Gbps links and further, while also maintaining the programmability that is expected of SDN based infrastructures. This performance is linked with the hardware that is chosen to serve as the base for the devices [8]:

[3] Defined as the throughput and latency of the network node

1. **General Purpose Processors (GPPs)** provide the greatest flexibility of all the solutions, while providing the worst results in performance, due to the general purpose design of the hardware and the optimizations present in the other architectures.

2. **Field-Programmable gate arrays (FPGA)** are a platform that enables the configuration of the devices via hardware design tools, maintaining the programmability of the GPPs, while also allowing for designing the devices around the tasks that they perform, including optimizations for switching/ routing. A notable example of platforms based in these systems is NetFPGA [4], an open source hardware platform designed for research, and supporting up to 100 Gbps operation.

3. **Application-specific integrated circuits (ASIC)** are integrated circuits that are customized for one particular application, removing the programmability, but also providing greater performance than the former options.

These architectures generally allows us to design SDN architectures around general purpose hardware, contributing to the flexibility of this paradigm, even considering the proprietary nature of the ASICs, which can be bundled with Software Development Kits (SDKs) for developing other applications on top of these.

Considering OF enabled hardware switches, the processing of incoming packets is done as by matching a (up to) 15 field tuple [9] to several flow tables, that have rules sent from the controller. In these cases, the possibility of bottlenecks is due to several factors, including the latency of the installation of new flow rules, and the memory limitation on the hardware. Solutions to the memory limitations in OF switches include DevoFlow [10], which utilizes wild card rules to reduce the number of flow entries that are installed on the devices, while also aggregating traffic, simplifying detection and management of unexpected large volumes of traffic (see section 3.1.2), due to reduced control plane load; and SmartTime [11] manages the timeouts for the rules on the switch, reducing this in the presence of micro flows, and increasing the timeout in the case of the occurrence of longer lived flows, which improves memory utilization and reduces the load on the controller.

Virtualised environments also allowed the development of Software Switches, due to the highly dynamic nature of virtual environments with frequent network topology changes caused by Virtual Machine (VM) movement between physical hosts. Furthermore, standard Linux bridges cannot handle the multi-server deployments [5] used in virtualised environments, which is why software switches like Open vSwitch (OVS) replace traditional switches in multi-server virtual deployements. OVS switches are also compliant with the OpenFlow protocol, which shows the flexibility that can be achieved by combining all the networking devices with one management protocol.

[4] https://netfpga.org/site/#/

[5] https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst

2.2.3 SDN Controllers

Central for operation of the networks, SDN controllers allow the orchestration of the multiple parts required for correctly operating a large scale network. Separate from the data traffic on the network, these are responsible for the interaction between the Northbound networking applications and the Southbound devices, as described in figure 2.8.

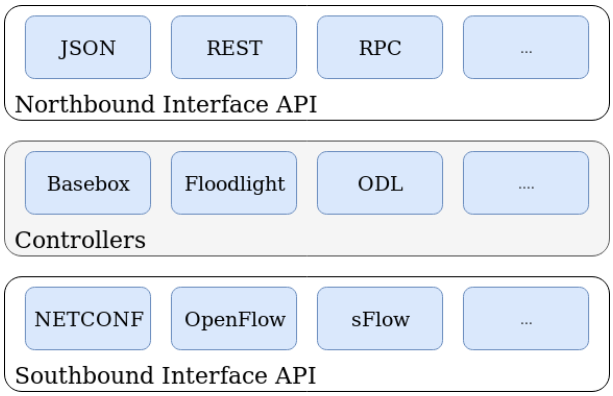


Figure 2.8: SDN Interfaces division

Although the use case of the controllers will depend on each deployment and implementation, the basic use case is to provide connectivity across layer 2 and 3 networks, which is achieved via flow management, including operations like switching, forwarding and potentially load balancing. The logically centralized position augments this capability by keeping the state of the entire network, which facilitates route planning and management.

Despite the advantages that the controllers centralization provides, this also introduces a Single Point of Failure (SPOF) [12], exposing a weakness to Denial-of-Service (DoS) attacks and controller failure. The potential catastrophic scenario related to controller downtime due to these failures means that an approach must be planned for disaster failure and recovery. High Availability setups are used to mitigate the potential failure of the controllers, by having multiple backups running. In order to guarantee that all controllers see the same network state, every switch must be connected to every controller, but as specified in section 2.2.3.6, only the Master controller writes the messages to the networking devices, which ensures that duplicate rules are not enforced. In case of controller failure, one of the backup controllers can take over the role of the previous master, without any outages on the network.

2.2.3.1 Existing Platforms

There are several controller implementations available for use, each with different interfaces, performance, and modularity. In [13] a comparative study is performed on available SDN controllers in 2014, and compare the different characteristics of each controller, like the available interfaces, the language of implementation, modularity, etc. In the following sections, we explore two of the highest rated solutions, Floodlight and OpenDaylight.

2.2.3.2 Floodlight

Floodlight [6] is a java-based SDN controller, and is one the first open-source solutions to gain relevance in research and industry [14]. The Northbound endpoint provides a REST API (see figure 2.9) to interact with the switch, which allows developers to get statistics, push flow entries, and more.

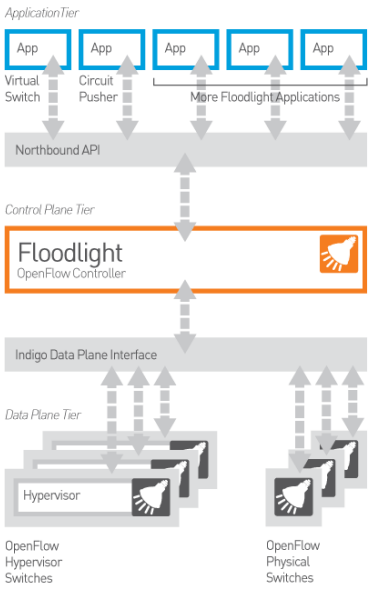


Figure 2.9: Floodlight architecture [15]

This controller also provides the OpenFlow interface, and it enables adding several modules, either through extensions, or through the utilization of the provided REST interface, simplifying the addition of new features to the base controller. It also provides an useful GUI for easier visualization of the topology, link state, and port statistics, which is visible in figure 2.10.

[6] http://www.projectfloodlight.org/floodlight/

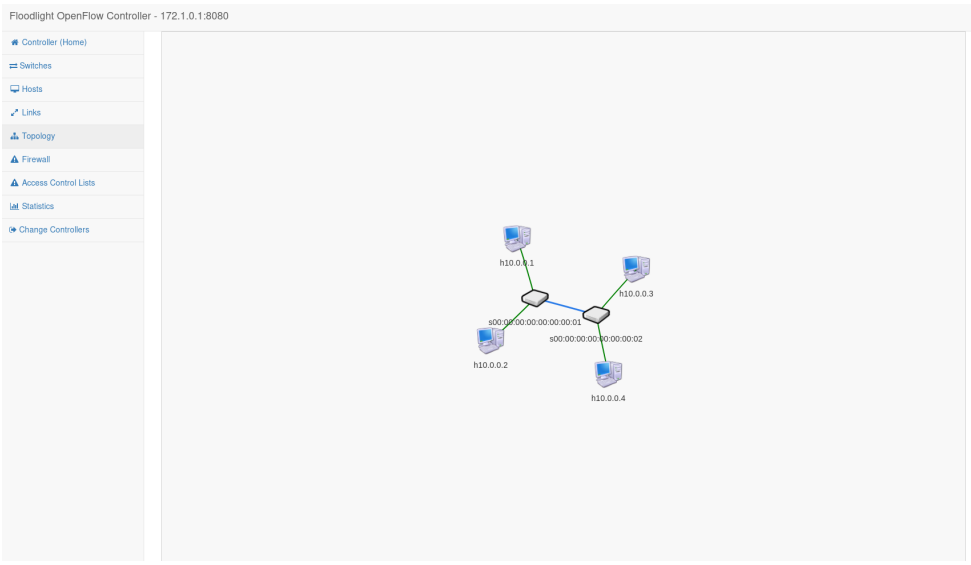


Figure 2.10: Floodlight web display of the topology

2.2.3.3 OpenDaylight

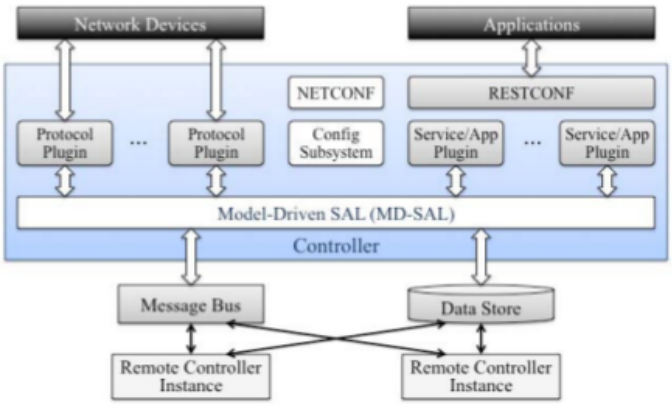


Figure 2.11: OpenDaylight controller architecture [16]

OpenDaylight [7] (ODL) is a project supported by the major vendors. The main differences between ODL and other controllers is support for other protocols in the southbound interface, due to the creation of a Service Abstraction Layer (SAL) [16]. In a high level overview, the creation of the Model-Driven SAL allows to extend the controller basic functionality by the addition of several plugins, which are used in combination with a RESTCONF [17] interface, defining the data models for the data stores, and the RPCs for interaction between the data.

[7] https://www.opendaylight.org/

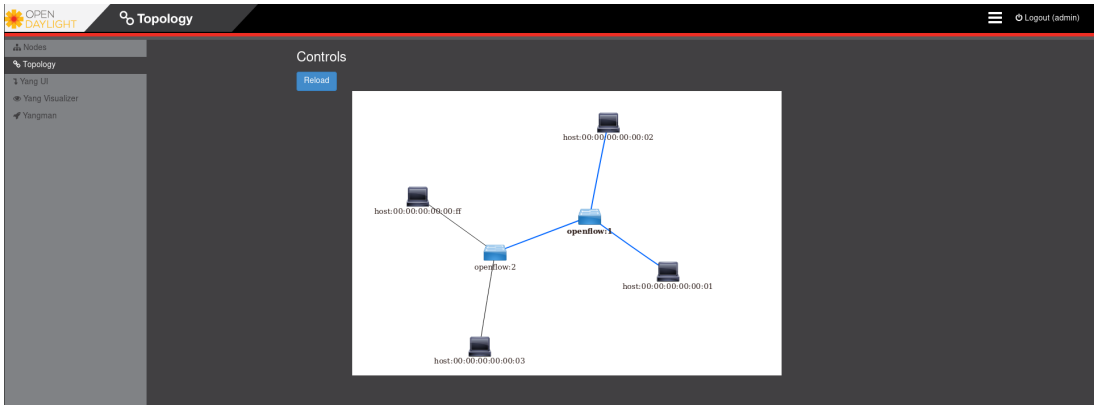


Figure 2.12: OpenDaylight Topology

2.2.3.4 Data center monitoring with SDN

The centralized view that SDN controllers maintain over the networks allows for it to keep the information about the flows currently present in the network. As such, the SDN paradigm allows for flexible control of the path the packets take in the network, and improves performance of the network at a large scale. By joining the information available on DCN and SDN, the requirements for traffic engineering (TE) in SDN, from the perspective of flow control are flow management, fault tolerance and traffic analysis [18]. This set of four requirements set the base for properly monitoring a DCN from the perspective of the SDN paradigm.

The next section are taken from [18].

2.2.3.5 Flow management

Flow management refers to the capability that the controller has to set rules for packet forwarding, and maintain the low overhead that is associated with registering a new flow rule, and also limiting the amount of flow entries, as hardware switches usually have a set amount of flow entries that they can support.

If we consider the fat-tree topology, one obvious consequence is the fact that if one controller is responsible for the management of the entire underlying topology, a bottleneck can be created when the rules need to be deployed to a node. When the switch receives a new packet, and there are no rules to properly forward this packet, then the packet is redirected to the controller, on the form of a PACKET_IN message, and after processing this packet, a new flow rule is sent to the switch. The problem with this scenario lies in the delay that it takes between the reception of the packet, and the installation of the new flow entry, which can be a contributing factor in packet losses in the data plane. This is an attack vector that is also explored in Distributed Denial of Service (DDoS) attacks for SDN platforms, as in an extreme scenario, the spoofed packet addresses will not have matches on the tables, which then result on overflowing the controller [19].

A solution for this issue is then related to decreasing the number of messages sent to the controller, by introducing some load balancing concepts. One of these concepts is related to the way that we can install

the flow entries on the switch. The information present in the packets serve to generate the flow-match entries that are deployed on the table. To reduce the number of interactions between the controller and OF switches, then we can reduce the number of match fields present in the flow rules, which reduces the number of flow entries on the switch and the controller messages. Another solution is distributing the controller among the network, but keeping them connected via a separate channel.

2.2.3.6 Fault tolerance

Although the switches are connected in a way that are able to mitigate link or other switch failures, in the case of faults occurring there needs to be the possibility of creation of new forwarding rules. An even bigger concern lies in the case when the controller fails, which will pose a larger problem in the network. For the case of node failure, fast recovery means that the OF controller can reactively act on link failures, by signaling the switches to forward packets toward new locations; or proactively, by setting the rules prior to the occurrence of the failure. In the case that the failure is short lived, then the controller is also responsible of resetting the paths to the optimal state. The way that controllers handle their connection is independent of the OpenFlow connection, and the failover should occur with minimal changes to the underlying flow rules and overhead.

2.2.3.7 Traffic analysis

So that the management tools can correctly display information about the state of the network, status statistics should be continuously collected and analysed. These statistics should provide the information about flows, packets and ports, so that the measured metrics can serve as a baseline for the decisions of the controller to adapt the flow rules to enable the best possible performance. Statistics can be collected in two possible ways: by continuously sampling packets from the switches; or applying sampling techniques and generalizing the information from the sampled data [20]. The problem here lies in the collection of the statistics poses a problem for large scale deployments, where continuously polling the network devices introduces both overhead and very large amounts of data to be parsed, or the data is not enough to detect failures in a short amount of time.

2.2.4 Statistics

2.2.4.1 OpenFlow

After exploring the requirements for network management, and the way the SDN model can support developing better systems, we now focus on the possibilities for obtaining this information from the networking devices. The OpenFlow protocol maintains a set of counters for each flow entry, port and group statistics, and this information can be queried to obtain a general view on the status of each OF switch. By sending specific controller-to-switch messages, the switch will return a set of the maintained statistics, which can then be parsed and analysed further.

Sending the port statistics message returns an array with the measured counters for each port. These counters include information like the amount of received and transmitted bytes/ packets, errors and dropped packets, and the duration that the port is alive.

The next important message is the OFMP_FLOW message, since this allows for getting the individual flow statistics, and obtain the information about each flow entry, including the time that it has been set on the switch, the number of packets/bytes in the flow, and the match fields. Also worth noting are the aggregate flow messages which describe how many packets are in the total flow entries, and also the number of flow entries that exist.

Also relevant is the information that is retrieved using the group statistics, as they allow to monitor the number of flows that direct to the group, and again the packet/ byte count that are matched with this group.

The information provided from these messages allows for a comprehensive view of the state of each switch, and a Network Management System (NMS) can utilize this information to achieve an understanding of the state of the network.

2.2.4.2 sFlow

One problem arises, however, when periodic requests generate too much information, and the control channel is overflowed with messages of port statistics, which is a possible scenario when the flow tables start getting too large. As such, a different alternative is to sample a small amount of packets from the switch, send the packet headers to the controller. One approach to this method is *sFlow*, a standard for collection, analysis and storage of network flows and traffic, for each device and its interfaces. sFlow is implemented using embedded agents on switches and routers, which compile interface and flow samples and exports them to the sFlow collector via datagrams.

Due to the problems that arise with continuously collecting traffic data, packet sampling has emerged as a valid solution to this problem, by collecting every n-th packet. The simplicity of the technique allows for reducing the complexity of sFlow agents, and having the sampling operation being done in hardware, resulting in the collection of the samples being done at the same speed of the channel it is monitoring. This reduces the losses that are inherent to the sampling process, which leads to biased analysis of the traffic [21].

Figure 2.13 shows the basic architecture that composes the sFlow system. One advantage of this system is the number of systems that incorporate sFlow agents [8], allowing for a detailed analysis of flows, and enabling flexibility for scalability in the network. By utilizing a sFlow collector that can accurately collect and process the datagrams incoming from the Agents, this protocol can be used to control most of the central aspects in network management, like troubleshooting network problems; controlling congestion on the network; or even analysing the possible security threats internal and external to the network.

[8] Complete list of compliant devices: http://www.sflow.org/products/network.php

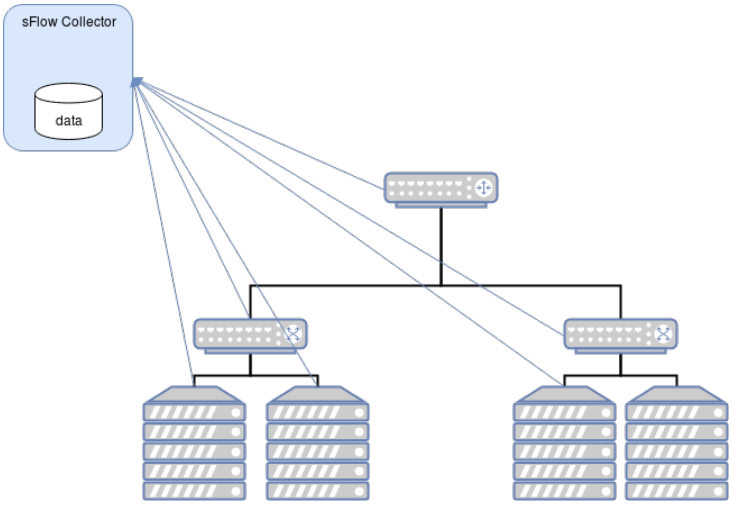


Figure 2.13: Architectural components of sFlow

2.3 BISDN

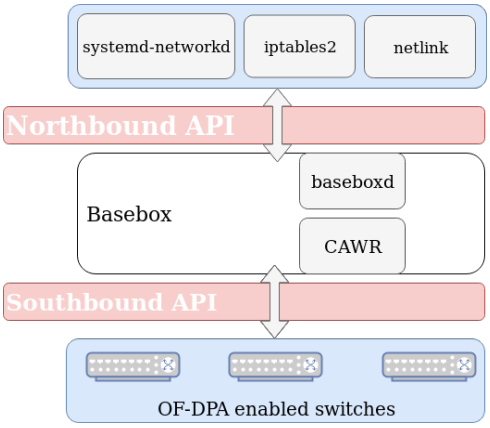


Figure 2.14: Basebox architecture

As the SDN market grows larger and larger in the networking world, new applications and products are developed and improved. Seeing the prevalence of closed source and proprietary solutions for this market, a need for open products that enable further growth and innovation in cloud DCNs is evident. The main gain in moving from vertically integrated solutions, is the decrease of costs involved, as cheaper solutions can be found in whitebox [9] switches and open sourced networking applications. With this motivation, BISDN developed Basebox, a Linux-powered solution to integrate switches and SDN controllers, allowing for data center operators to configure and manage networks using linux commands, removing the need for having to manage several devices with different interfaces and workflows, and adding the capability of running standard networking applications on top of the controllers and switches.

[9] whitebox switches are generic switches that possess no association to a certain vendor

Basebox also includes the possiblity of running in a failover scenario, by introducing a backup controller for the network, and the possibility of creating a giant switch abstraction, by adding another controller, Capability AWare Routing (CAWR), and having this manage all the southbound switches.

2.3.1 Existing product

2.3.1.1 baseboxd

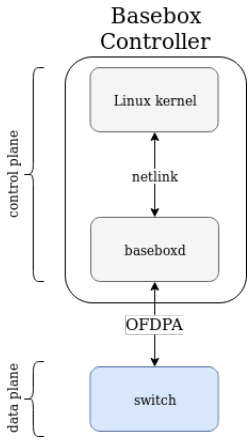


Figure 2.15: Diagram displaying baseboxd's capabilities[28]

baseboxd is a controller daemon connecting whitebox switches with a Linux-based system. The controller communicates with the Linux kernel over netlink [10] and with the switch using the OpenFlow Data Path Abstraction (OF-DPA) which will represent the state of the switching infrastructure as part of the Linux networking stack.

From the perspective of the switch, the baseboxd listens for OFPT_PORT_STATUS async messages, and updates the state of the interfaces in the Linux kernel, creating tap interfaces for each port that is up, and deleting them when they go down. On the controller side, the changes done to the interfaces are also propagated downwards into the switch, for example the addition of VLANs, or setting route to neighbors, and sends the appropriate flow messages via the OpenFlow protocol. Since baseboxd responds directly to the relevant netlink messages, it is one of the intended ways to interface with baseboxd. One may use tools such as iproute2 and systemd-networkd to configure baseboxd through this interface [28].

2.3.1.2 CAWR

CAWR is a secondary OpenFlow controller that creates a giant switch abstraction from a set of whitebox switches. This giant switch integrates with baseboxd and allows for easy scaling and management of

[10] netlink is an Inter-Process Communication's (IPC) socket for exchanging information between the Linux kernel and userspace applications. This interface provides a communication framework to configure a network's control plane.

multiple networking devices. As a secondary controller, it is placed in between baseboxd and the physical switches, and both its northbound and southbound interfaces are OpenFlow, following the OF-DPA standard.

CAWR was designed to handle multi-switch configurations, and the current version supports up to two switches. A host can be connected to each of the switches by a pair of interfaces that have bond [11] mode configured, and the layer 2 traffic across the physical network is properly forwarded. Failover mechanisms to deliver uninterrupted operation are present, in the case of port or switch failure.

CAWR adds Link-Aggregation Control Procotol (LACP) (IEEE 802.3ad, IEEE 802.1ax) and Link-layer Discovery Protocol (LLDP) -based (IEEE 802.1ab) topology discovery to the Basebox setup. CAWR uses LLDP to detect internal links (connections between the switches) to build an initial topology, and bonds that are discovered via LACP or have been manually configured are added to the topology as well. LACP is also used to continuously monitor the link status and detect port connections and disconnections on servers that have LACP enabled [28].

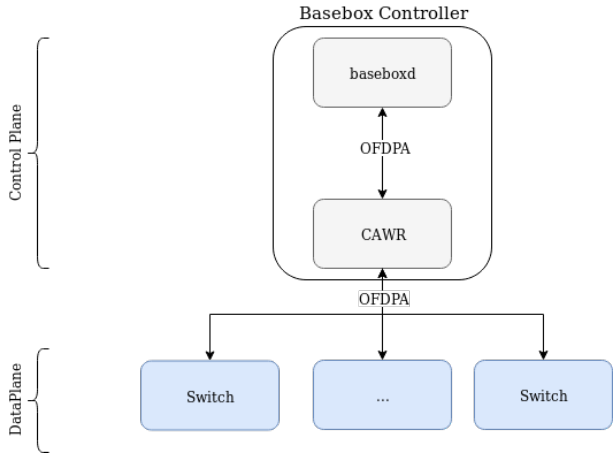


Figure 2.16: Diagram displaying CAWR's capabilities[28]

[11] Network Interface (NIC) bonding is a process of combining two separate network interfaces into a single interface. This technique guarantees performance increase and redundancy.

Related Work

3.1 Traffic engineering in data centers

Understanding the impact of elephant flows in the normal operation of a data center requires understanding the traffic characteristics of the typical cloud data center. The geographical proximity and localization of large data centers optimizes the interoperability that applications may require by minimizing propagation delay that could be present if the links between servers, however cloud data centers used for costumer faced applications and those employed in data intensive tasks may present different requirements, which poses a problem in optimizing the underlying network. Furthermore, absence of publicly available data contributes to the challenge of researching data centers [37].

Typical cloud data centers operate at a ratio of 1:1000 staff members to servers [34], which points to an essential need for extensible automation and failure recovery plans for optimal operation. Automation is central for cost reduction strategies in data centers, reducing the impact of failures caused by human errors.

Cost management is also achieved by improving power consumption, which correlates with improved methods for balancing load on the servers [42]. **Load balancing** is the concept of moving the load of an overloaded server to an underutilized one, which reduces performance degradation, and increase recovery from failures [40]. In a virtualised environment the possibility of moving Virtual Machines (VMs) across servers and racks facilitates distributing the load without downtime, but the migration decision is not trivial due to the large amount of variables involved, for example, the bandwidth, memory and CPU that is available on each server supporting the migrated VM. Furthermore, if an application's workload changes over time, one decision may not apply for further migrations [35]. Netshare [36] proposes a system that optimizes bandwidth allocation by imposing max-min fair sharing on services, using a centralized controller for orchestration. In [31], a minimization approach is applied to the migration problem, by generating a cost function considering the variables associated to VM placement, computing the impact of moving a certain VM to a physical host, and the migration destination is selected with the least amount of generated impact.

Caching is the act of duplicating content across a network, in order to optimize access to frequently accessed content, minimizing network congestion at peak access hours [33]. For Software Defined Net-

working, the centralized controller provides an optimal environment to implement caching in data centers due to high level knowledge of the tenants in the network. Moirai [39] presents a programmable data-plane caching system, that allows to prioritize workloads, providing per-workload bandwidth guarantees. In [32], the last mile delivery of Video-On- Demand problem is solved with the inclusion of OpenFlow to create dynamic caches using hardware independent statistics, and provide support for additional policies like load balancing. These systems show an evolution from historical caching mechanisms, that do not tend to support operations necessary in data center network operations, with higher bandwidth and storage requirements [39].

3.1.1 DCN Traffic

The techniques proposed for traffic engineering in traditional networks do need to be revised in DCN's, since metrics like propagation delay can be negligible, due to the physical proximity of nodes in DC's [57]. However, research in this topic can be a difficult task, since many data center operators do not publish information about their applications and services.

By collecting data from different types of DC's, several studies have been made about the traffic characteristics [37, 59, 58]:

- The placement of VMs and servers effects the bandwidth and link capacity, due to the variety of applications that can be running on the servers at any time, and this non-uniform placement of VMs contributes to higher amounts of traffic originating from the same rack
- The majority of flows [1] are described as being small in size, and short in duration, which are usually described as *mice* flows. The counterpart to these are the *elephant* flows, which occupy a very large share of the bandwidth, and degrade application performance, due to a choking effect to the latency-sensitive mice flows. Applications are tied to the type of traffic they generate, where online gaming, VoIP and multimedia broadcasting usually originate mice flows, where the large data transfers and file-sharing generate elephant flows. Despite 90% flows are small and last hundreds of milliseconds, total traffic volume is largely dominated by the remainder, called **elephant flows** [37]
- In a normal situation, link utilization is low in the layers apart from the core switches. In addition to this discovery, losses are not associated with spikes in traffic, instead being related to high utilization of the link, which is one of the effects of the previously mentioned elephant flows.

Software Defined Networking based monitoring allows to increase the capability of conducting traffic monitoring and measurements. OpenTM [38] utilizes the monitoring information to build the Traffic Matrix (TM) of an OpenFlow network by employing different methods of querying the switches, allowing to reduce the load associated with these queries and taking into account the different processing

[1] flows are sequences of packets sent from a source to a certain destination, either host, anycast or multicast domain

power of each device. Another method for estimating the link utilization is present in FlowSense [44], in which PACKET_IN and FLOW_REMOVED OpenFlow messages are monitored since these messages carry information of arrival of new flows and expiration of flow entries, providing a zero-overhead monitoring technique. This method has optimal performance when the flows in the network are short lived, posing worse performance as the flow time increases. Due to the absence of these messages in longer lived flows, the performance decreases when applying this technique on elephant flows.

3.1.2 Elephant flows

Detection of network anomalies is subject to intense research, and as such, several methods were developed, that assume different levels of control over the network and provide different results to different use cases. In this section we focus on the available techniques for detection and mitigation of large flows.

3.1.2.1 General formulation of elephant flows

Networks can be represented as an undirected graph $G(V,E)$, which contains a certain multicast group $M \subseteq V$, and a multicast tree $T \subseteq E$ [49]. Considering the nodes $u,v \in M$, and the unique paths $p_{u \to v}$, an approach to define elephant flows is to consider the set of flows that optimize the Quality-of-Service (QoS) on the unicast path p, or the multicast tree T such as in [49, 56].

Considering the set of flows F on the path p,

$$F =< F_1, F_2, ..., F_n >,$$

and an arbitrary Quality of Service function $q()$, then $x_p = q(F)$ is the QoS requirements for path p. In [47], the optimal partition is the configuration that improves the QoS in p, or, in a formal definition:

$$q(F_1', F_2', ...F_n') > q(F_1, F_2, ..., F_n),$$

where F_X' is an optimized set of flows on path p. $q()$ is a generic function, since QoS can be classified with several parameters, like bandwidth, latency, error rate, and so on, which must then be adapted from case to case. These parameters can be classified in two classes: *bottleneck* and *additive* [49]. As previously mentioned, the main effect of elephant flows is the choking effect of the bandwidth of the shorted lived flows, making the bandwidth a proper metric for usage in $q()$. Considering bandwidth as the global QoS requirement Q, F_p' must satisfy the condition

$$x_p' = Q_{F \in p},$$

since bandwidth can be classified as a bottleneck parameter, where the link with the lowest bandwidth sets the total for the link. Considering the additive case, where the delay of the links can be used as an example, the total delay of the links will be the sum of the delays of each link.

Using this notation, $< F'_e, F'_m >$ identifies the optimal QoS partition where F_e is the set of elephant flows, and F_m is the set containing mouse flows. From this framework, the set of elephant flows F'_e is

$$F'_e =< F'_1, ..., F'_X > \subset F,$$

that correspond to the smallest set of of flows that maximize the QoS function for a set of links. Formally, this restriction can be defined as

$$\begin{aligned} &min(X), \\ &max(q(F'_e)) < Q. \end{aligned}$$

3.1.2.2 Elephant flow detection

The problem of detection of traffic anomalies has been subject to extensive research, and several different approaches have been proposed. These methods are based on different techniques [20]:

- Modifications to the applications and services to notify the controller about the state of the traffic on each service. Despite this approach resulting in the most accurate "detection" of network anomalies, support for this technique is not extensive, due to the required changes to each service, and it does not account for abrupt changes on the traffic
- By setting hard limits on the transmission capabilities of each port and switch, and enforcing via shaping mechanisms, the controller is ensured of the non existence of flows that could impact network performance. This is a mitigation strategy that does not scale well to very large networks, since it requires the storage of the rules imposed to every port, and can potentially lead to the inefficient use of network resources, reducing the flexibillity on the DCN's.
- By employing sampling and collection techniques, using mechanisms like sFlow (see 2.2.4.2), and building the profile of the normal state of the network, this method can detect outliers that deviate from the normal state of the network. Utilizing this method reduces the impact of continuously polling the network, while reducing impact on the packet and byte counts [21], but the loss of information inherent to sampling may be a challenge to successful deployments, which should account for optimal sampling strategies and inference from the obtained statistics.
- Periodic polling of the statistics from the switch, and employing statistical analysis methods to determine change points in the state of the network.

Mahout [20] presents a system that allows for elephant flow detection by monitoring end hosts. Via implementation of a shim layer on top of every host present in the network, the proposed system allows to tag the traffic that belongs to larger flows, reducing the complexity that is generated if the monitoring were done at the aggregation or core switches. Detection itself is done by comparing the number of bytes

in a buffer to a pre-defined threshold. In Hedera [45] the problem of maximizing network bandwidth is via detection of large flows and optimization of the placement of the switches according to the demands of these flows. A sampling approach for detection of network anomalies is explored in [43], where the proposed method relies on analysing the sampled packets and building a tuple of the traffic characteristics (source and destination IP, source and destination ports, and the protocol). Using machine learning methods, this system then compares the entropy of each field to a pre-defined threshold, which allows for detection and classification of different types of DDoS attacks. In TE, the use of entropy has been vastly discussed [21, 41], since it can be used in fine grained traffic engineering systems, as well as anomaly detection systems, since entropy based methods detect the changes in traffic distribution, and changes that would be too small for volume change detectors are visible in these systems. Furthermore, traffic classification is now possible, since similar changes in traffic distribution cause the same changes in entropy, and this factor can be used in classifying the different types of DDoS attacks, for example.

In the case of systems that explore the changes in packet and bytes, [30] proposes a simple approach for traffic change detection, based on time series and Principal component analysis (PCA). By measuring the state of the network, using the links, flows and packets statistics, a statistical approach to finding the time locations of change points in the network traffic is designed, by combining different types of statistical process control techniques. PCA removes the correlation of a set of observations, separating the seasonal variation from the residual variability. Another system exploring PCA for anomaly detection is studied in [48], where traffic flow measurements from Origin-Destination (OD) links are used to determine the anomalies present in the link.

3.1.2.3 Elephant flow mitigation

Now that the methods for identifying elephant flows were presented, discussion of mitigation techniques to decrease the impact on the smaller, latency sensitive flows must be addressed.

In general, post-detection actions for solving elephant flows can be based in [51, 50]

- separating the queues for elephants and mice flows,
- routing the elephants on a separate path or forwarding them in separate networks,
- splitting the elephants into smaller flows, using different ports and relying on reliable transmission mechanisms, such as those present on TCP to organize and re-order the packets.

A novel approach for mitigation of elephant flows is present in [52], where the timeout present in OpenFlow rules in presented as a dynamic way to manage the control plane load and improve the usage of flow table resources, managing the flows based in their inter packet arrival interval and periodicity. Basic functionality of this system is the management of the timeout setting of each flow, since, if this is proven to be too long for flows with long inter packet arrival periods, the flow rules on the switch could become out dated, leading to a possible switch table overflow; and if the timeouts are too short, the control plane can be overloaded, causing possible controller failures.

Another solution proposed for managing elephant flows are present in congestion-aware systems, where the routing or forwarding decisions are done with a dynamic overview of each link properties. The proposals in congestion-aware systems are relevant, due to the inefficiency of Equal-cost multi-path (ECMP) routing in data centers, which causes low throughput and bandwidth utilization for elephant flows, and increases latency for mice flows [53]. In [54], link capacity is managed per flow by analysing the *flow deadline* [2]. The reasoning behind analysis of the flow deadline is due to splitting resources on the link. If the link's capacity is shared equally between every flow, then every flow will arrive at the same time, but likely the deadline will have been missed by some. Furthermore, consideration of the flow implies that every packet must arrive before the deadline. In the case of elephant flow mitigation, assigning larger portions of the link's capacity to the shorter flows can be a possible strategy. A similar approach is present in [55], instead analysing Round-Trip Time (RTT), since this metric directly reflects the end-to-end latency of a link.

3.2 Time series analysis

As a result of the large scale of current data centers, maintaining control over these networks proves a difficult task. Networks operators must then adapt to the current situation by improving the monitoring infrastructures to allow faster response to problems, and **Root-Cause Analysis** (RCA) [3] of the source of network issues can be done faster and easier, which will reflect on better service and lower costs for network operators.

Network behaviour analysis is defined by the constant monitoring of a network, so that events that compromise the "healthy" state of the network can be removed or mitigated. These include not only cases where the anomalies are caused with malicious intent, like the case of DDoS attacks, but also failure of network devices or changes in user behaviour [30]. These systems are equipped with alarm capabilities, so that system administrators can quickly respond to changes, possibly even giving some information about the source of the problem. However, the automation of these monitoring processes means that the possible existence of false alarms reduces the operators capabilities to act on actual failures.

Understanding processes and their results is a key factor in the success of implementing new features, or analysing existing ones for their efficiency and output. This analysis, important for the different fields in engineering, economics, health, allows obtaining information about the normal and abnormal state of each underlying process, provide forecasts and predictions on short and long term behaviour of the relevant data, classify and cluster information, and more. The act of collecting and processing the data, over a period of time is called *time series analysis*.

Monitoring systems provide a guarantee in quality engineering, since they allow to follow a system and its properties, and notify operators if changes happen that impact the normal status of a relevant parameter. These changes can be occasional or systematic, but should the state of the system deviate from

[2] Flow deadline is defined as the time that the flow has to finish transmission, usually due to Service Level Agreements (SLAs)

[3] method of identifying the initiating cause of an error or fault in a system

the limits set by the operators, researching the cause of the errors can reveal some errors or malfunctions in the system. *Change detection* is the study of the different parameters of the system, and determining the points where these cause a significant deviation from the normal operation. In network traffic analysis, these methods can be applied to determine when the behaviour of certain flows, that can be monitored over metrics originating from the controller and switches, impact the traffic characteristics. One key part on the application of changepoint detection is the understanding and selection of the proper metrics to monitor, to ensure that these are sensitive to the traffic changes. One other important consideration in applying changepoint detection mechanisms is the reduction of false alarms, that occur when the metrics are too sensitive to traffic changes, and limit the network operators capability of accurately responding to real network issues.

Change detection mechanisms are classified as follow [30]:

- **Online vs Offline** In change detection theory, an important distinction is the difference between online and offline detection. Offline, or batch detection methods consider a fixed length of observations, and retrospectively analyse the dataset to determine the time where the change, or changes took place. Online detection, or sequential detection, unlike batch detection which uses all the available observations to detect the changes, including the ones obtained after the change took place, is based on the determination of the change points based on the arrival of the new data, allowing for determining the change as fast as possible [60].

- **Parametric vs Non-parametric**Another important distinction is related to the scalability, and the amount of data that needs to be stored to accurately implement detection on new samples. Parametric approaches rely on learning a probability distribution from the monitored variables, and using this learned data to estimate the unknown parameters, after which the training data can be discarded. Non-parametric models however, do not take into consideration the distribution of the monitored variables, and analyse statistical properties instead. The cost of this analysis is that the previous data must be stored to provide better results, but this problem can be mitigated using algorithms like sliding windows or moving averages.

3.2.1 Mathematical formulations

A time series can be defined as a stream of observations $X = \{x_1, ...x_i\}$, where x_i is a vector arriving at time i. The time series X can also be described as the sum (in equation 3.1), or product (in equation 3.2) of the following components: S_t, which refers to the seasonal component of the data; T_t, which defines the trend of the data, and R_t represents the residual values, accounting for uexpected variation and noise.

$$X = S_t + T_t + R_t, \tag{3.1}$$

Or:

$$X = S_t * T_t * R_t \tag{3.2}$$

In regards to the classification of the trends according to the type of change, they can be classified as:

- **Deterministic** when the trend is consistently increasing/decreasing
- **Stochastic** when the opposite happens

Time series data usually present a non stationary behaviour, that is characterized by changes in the mean and variance. Statistical methods require, however, stationarity in the data. The presence of trends and cyclic behaviours is the most common violation of stationary, and table 3.1 shows the most common trends present in the data, and the parameters that need to be learned from the time series in study.

Linear	$y = m \cdot x + b$
Polynomial	$y = b + c_1 \cdot x + \ldots + c_n \cdot x^n$
Exponential	$y = c \cdot x^b$
Logarithmic	$y = a \cdot ln(x) + b$

Table 3.1: Trend models

Removing systematic changes like trends is also possible with differencing,

$$\nabla X_t = X_t - X_{t-1}.$$

3.2.2 Forecasting

Time-series' have the possibility of applying statistical models to extract the next value prediction based on past observations. Under the assumption that the underlying process can be modeled by previous historical values, and assuming this model remains true for future measurements, the time series data historical behaviour can be used to generate these forecasts for future values in the time series.

3.2.2.1 Exponential smoothing

Exponential smoothing allows for generating predictions using the historical behaviour, by applying a set of weights to the data that exponentially decreases over time. Considering the time series X_t, the one-step ahead prediction $\hat{x}_t$ can be obtained by equation 3.3. In this model, the smoothing factor $\alpha (0 < \alpha < 1)$ should be obtained empirically, and its value will determine the forgetting factor for the past observations.

$$\begin{aligned} \hat{x}_1 &= x_0, \\ \hat{x}_t &= \alpha x_t + (1-\alpha)\hat{x}_{t-1}, t > 1. \end{aligned} \tag{3.3}$$

Due to its simplicity, this method is not suitable for situations where the data has trends, or seasonal behaviours [62]. The solution for this problem is introduced with double exponential smoothing, also know as Holt forecasting, and triple exponential smoothing, also known as Holt-Winters forecasting.

These methods introduce further components to dampen the effects of cyclic behaviours in the data. Double exponential smoothing is defined by [30] :

$$\begin{aligned}\hat{x}_t &= L_{t-1} + T_{t-1},\\ L_t &= \alpha x_t + (1-\alpha)(L_{t-1} + T_{t-1}),\\ T_t &= \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1}.\end{aligned}$$

And triple exponential smoothing is defined by:

$$\begin{aligned}\hat{x}_t &= L_{t-1} + T_{t-1} + I_{t-1},\\ L_t &= \alpha(x_t - I_{t-s}) + (1-\alpha)(L_{t-1} + T_{t-1}),\\ T_t &= \beta(L_t - L_{t-1}) + (1-\beta)T_{t-1},\\ I_t &= \gamma(x_t - L_t) + (1-\gamma)I_{t-s}.\end{aligned}$$

The components these methods introduce account for the cyclic behaviours in the data: L_t accounts for the baseline behaviour of the data, which is calculated on the simple method; T_t smooths the trend with the β parameter; and S_t accounts for the seasonal components with the γ parameter. As with α these parameters should be defined mostly by previous experience.

3.2.2.2 Autoregressive Moving average

Approximating the time series data to a model allows for generating predictions for the next values, since the relationship between two variables in the data is then known. The Box-Jenkins method defines the steps of building this model as [61]:

1. **Identification** Model the data, by reducing the variables to a stationary state, and removing the possible seasonality in the series
2. **Fitting** Estimate the parameters for the model
3. **Checking** Verify if the model accurately fits the available data, returning to the identification step if its not adequate

Modelling the time series data is possible through moving average (MA), autoregression (AR) or a mix of the two (ARMA) processes. The autoregressive model, also referred to as AR(p), considers the linear regression of the p past values, as equation 3.4 shows.

$$x_t = \sum_{i=1}^{p} \phi_i X_{t-i} + \varepsilon_t, \tag{3.4}$$

where ε_t is the white noise component, a purely random process of mean 0 and variance σ^2, ϕ_i are the constant parameters of the model, and p defines the order of the model.

The moving average process models the time series to the white noise that has occurred in the previous periods, as shown in equation 3.5. In this equation, θ_i are the parameters of the model, μ is the mean of the series, and ε_t are the white noise components. Similarly as in the autoregressive model, the moving average process can be defined as MA(q), where q is the order of this model.

$$x_t = \sum_{i=1}^{q} \theta_i \varepsilon_i + \mu, \tag{3.5}$$

Finally, the ARMA(p, q) process is defined as the combination of both methods, as shown in equation 3.6.

$$x_t = \mu + \sum_{i=1}^{q} \theta_i \varepsilon_i + \sum_{i=1}^{p} \phi_i X_{t-i}. \tag{3.6}$$

Finding the p and q parameters of the models is done via observation of autocorrelation and partial autocorrelation functions, as introduced in the Box-Jenkins method [61]. The autocorrelation function calculates the correlation of a time series with its own lagged values. The behaviour of this function can provide us with the information on the these parameters: if the sample autocorrelation shows an exponential decrease then the process can be modelled with an AR method; and if this function shows a drop after a certain value q, then the moving average model is better suited for modelling the time series [30].

3.2.2.3 Error calculation

A very important part of building forecasting modules is assessing the associated errors with the prediction. Minimizing prediction errors improves the quality of the forecasts, via the adjustment of the chosen parameters for the model.

A common way to calculate the error is via the Squared Sum of Errors (SSE), which is shown in equation 3.7.

$$SSE = \sum_{t=1}^{T} (x_t - \hat{x}_t)^2 = \sum_{t=1}^{T} \varepsilon_t^2 \tag{3.7}$$

Other methods include the Root Mean Squared Error (RMSE),

$$RMSE = \sqrt{\sum_{t=1}^{T} \frac{(x_t - \hat{x}_t)^2}{n}}$$

or the Mean Absolute Percentage Error (MAPE),

$$MAPE = \frac{100}{n} \sum_{t=1}^{T} \left| \frac{x_t - \hat{x}_t}{x_t} \right|,$$

In [72] a comparative study of these measures is presented to determine the most adequate measure for univariate time series forecasts. They analyse simple forecasting methods like the exponential smoothing and Holt's method, and the out-of-sample [4] and in-sample [5] performance of these methods. This analysis of the different forecasting accuracy measures is finished by concluding that the Mean Absolute Scaled Error (MASE), seen in equation 3.8 due to simple interpretations, and independent of the scale of the data, which poses problems when comparing forecasts with different scales. In equation 3.8, the numerator e_t is the forecast error for a certain period, or $e_t = x_t - \hat{x}_t$.

$$MASE = \frac{1}{T}\sum_{t=1}^{T}\left(\frac{|e_t|}{\frac{1}{T-1}\sum_{t=2}^{T}|x_t - x_{t-1}|}\right) \tag{3.8}$$

3.2.2.4 Change Detection

A relevant indicator for the validity of the generated forecasts is the forecast error, that is calculated via the difference between the real measurements and the predicted value. Furthermore, by assuming a distribution for the forecast error, and a certain significance level, it is possible to validate the generated forecasts, and detect values that do not fit the model, accusing a possible variation in the parameters of the model. As such, the prediction error is able to be employed in change detection algorithms. This prediction error can be defined as:

$$\varepsilon_t = x_t - \hat{x}_t$$

Hypothesis testing is used to perform a test of an assumption about two random variables. This hypothesis states that a certain relationship between the two variables exists with a certain significance value, and this relationship can be a relation in the means, the distribution of the observations, etc. The first step is set **null hypothesis**, which is the desired assumption to test, and is referred to as H_0, which is then tested against the **alternative hypothesis**, H_1, which is considered true if H_0 is rejected. The validity of H_0 is based on a comparison between the two data sets, according to a certain threshold, called the significance level. This significance level also defines the probability of wrongly rejecting or accepting a hypothesis, which are defined as following errors: **Type I error**, happening when the null hypothesis is rejected, but the performed test is true, and the **Type II errors**, occurring when the opposite happens.

Originating from quality engineering, **control charts** are common ways of following the output of a certain process, with the aim of reducing variability associated to manufacturing processes. Control charts allow to evaluate the possible sources of variation, and classify the output of the process, based on the mean or variation of the sampled process, as **in control** or **out of control**, depending on the causes of variation. The process is considered in control when the parameters of the monitored variable, like μ_0 or σ^2 are inside the predefined **control limits**, that are usually set as requirements for the process. The wide range of control charts allow for a flexibility in choosing the right one that fits each application. Common

[4] used for evaluating the forecasting accuracy, by using samples not belonging to the initial fitting period
[5] using data available on the initial fitting period to generate one step ahead forecasts

charts used are those proposed by Shewhart, where the measurements from the process in study provide a statistic, like the mean, range, etc. Plotting these parameters allows for drawing the center line at H_0, and the control limits are defined by a multiple of the standard deviation. This control chart allows for actions to the process be performed not only when the points are shown to be out of control, but also when there is a sequence of values above or below the center line, or a upward or downward trend is shown in the control charts.In the context of change detection, the hypothesis test relies on H_0 stating that there is no change in the parameters of the sample like the mean or the variation, and the second hypothesis H_1 stating the contrary.

The CUSUM (**cu**mulative **su**m) control chart provides a test based on *stopping rules*, where the alarms are raised when a parameter of the distribution θ_t, like the mean, or the variance exceeds certain thresholds. In the parametric case, the CUSUM algorithm for detection of a change at t_0 from the observation x_i is based on the log likelihood ratio defined by [60].

$$S_t = \sum_{i=1}^{k} s_i = \sum_{i=1}^{k} \ln \frac{P_{\theta_1}(x_i)}{P_{\theta_0}(x_i)}.$$

The previous equation is related to the negative drift of S_t under normal conditions, and the positive drift after a change is detected. The alarm is raised when a test statistic g_t is larger than a threshold h, and can be obtained by

$$g_t = S_t - \min_{1 \leq i \leq t} Si \geq h$$

In the case when $P_\theta(x)$ is not known, the log likelihood parameter cannot be calculated, a non-parametric approach must be used, as mentioned in [60].

3.2.2.5 Performance Evaluation

A relevant metric for designing change detection models for application under network traffic is the *false alarm rate*, due to the impact that incorrectly identifying the normal state of the network as abnormal can have on the operators capability of addressing real abnormalities. This statistical difference of errors present in hypothesis testing is not relevant for network operators [30], which implies that the design of the change detection mechanisms should reduce the total false alarm rate. The effectiveness of a statistical approach for change detection can be seen as the relationship between false and real alarms, since the trade-off with online detection is the number of falsely raised alarms, and the number of accurately reported changes. As mentioned in section 3.2.2.4 the possible wrong decisions are the type I and II errors, which are usually represented with error, or confusion matrices, seen in table 3.2, where the possible outcomes from the decision algorithm are displayed. The notation for this table is True Positive (T_P), False Positive (F_P), False Negative (F_N) and True Negative (T_N).

		Predicted	
		H_0	H_1
Actual	H_0	T_P	F_P
	H_1	F_N	T_N

Table 3.2: Generalized confusion matrix for hypothesis testing

In the following equations, we define the ratios that allow for measuring the performance of the algorithm, with Accuracy (A), measuring the correct decision of the algorithm; Sensitivity (S), indicating the capability of the system to detect the change, and Precision (P) indicates the ability of the algorithm to accurately distinguish between true and false alarms.

$$A = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} = \frac{T_P + T_N}{N_{alarms}}$$
$$S = \frac{T_P}{T_P + F_N}$$
$$P = \frac{T_P}{T_P + F_P}$$

The performance of the change detection algorithms can also be defined by:

- MTFA (Mean Time Between False Alarms)
- MTD (Mean Time to Detection)
- ARL (Average Run Length)

The ARL is the expected number of samples required before an alarm is provided, and it can be divided further in two important measures: ARL_0, which specifies the expected number of required samples until the alarm is raised, assuming that the process is in control; and the ARL_1, indicating the expected number of samples until an alarm is raised, under the condition that the process is out of control. For optimal results for change detection, we require that ARL_0 is as large as possible, and ARL_1 be as small as possible.

Monitoring SDN Switches

4.1 Problem

Developing solutions for use in mission critical environments requires the deep understanding and analysis of the requirements of these environments, like those present in large data centers. The Basebox system (see section 2.3) is an example of these solutions, but lacks a system for monitoring and management of the network devices. Such a system should

- display the topology information reported by CAWR, including the internal switch links, and the LACP discovered bond interfaces on the servers,
- display the port and link statistics for both switches,
- design an alerting system, so that network operators can be informed of changes on the network state,
- provide diagnostic capabilities.

Also addressed by this system should be the definition of Quality-of-Service (QoS) policies, that maintains the levels of accepted traffic behaviour of each device in the network, and identifies and applies some automatic mitigation strategy when the system does not perform according to normal state. By understanding data center traffic characteristics, one of the largest problems are the existence of *elephant flows*, that impact the available bandwidth of the network. As such, the development of a full management system should also include a system that receives the incoming port statistics, analyses these, and applies statistical analysis to manage the impact caused by elephant flows.

4.2 Solution

Following the previously presented requirements, the proposed system is present in figure 4.1. First, we add an interface for the controllers that exposes management information like the network's topology, statistics, etc. After this, we build an **Operations Support System** (OSS) that provides the basis for the development of applications that monitor the state of the network, using the implemented API to

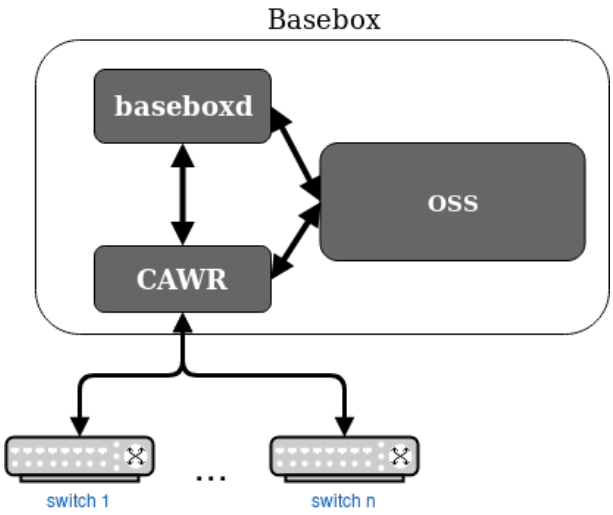


Figure 4.1: High-level visual description of the proposed system

obtain the relevant information. This system allows further separation of roles in the network, in contrast to a system where the controller would gather the roles of managing and monitor the network's status, increasing the load on the controllers. Furthermore, this architecture increases the modularity of the components, enabling hot-swapping different modules, and allows parallel development of different features in the monitoring and management stack.

We provide a proof-of-concept composed of two components: the first is a Graphical User Interface (GUI) providing an user friendly interface to display topology and statistics, and the second is an intelligent system enabling the detection of elephant flows. In this section we describe in a high level way the approaches for the development of this system.

4.2.1 GUI

The primary use case for this component is following the changes in the underlying topology, while also allowing the monitoring some aspects of the port statistics, and as such, the links between switches and the hosts, and the association between the ports and the switches should be displayed. Performance wise, this system must run as fast as possible, and the transmission of data must not interfere with the systems operation.

In order to reduce the memory requirements of the platform, and decrease the time that it takes for drawing topology updates, a decision was made to not store state, which would increase the time it takes for topology updates, with queries to store and load data from these databases. This also removes complexity as the system grows, where storing information about links and switches would dramatically increase database size.

Motivated by compatibility with standardized systems, choosing the data model for representing the underlying system required the investigation of similar systems, and the RFCs, or similar standardized documents that exist.

4.2.2 Elephant Flow detection

The second component of our system was developed for monitoring the system statistics, and displaying alerts when elephant flows are detected. For this end, we have implemented an algorithm for the detection part.

By monitoring traffic changes in the switches ports, the developed system aims to detect the presence of large flows based on the traffic statistics obtained from the API. As we assume the tree topology in data centers, seen in figure 2.2, the testing environment must be designed to monitor the lower layer edge switches, connected directly to the physical hosts. By monitoring the ports on these switches, we detect changes in the reported traffic statistics via a developed Python script.

Management API

Basebox is a SDN controller used in data centers, which are a mission critical component for network operators. As such, management capabilities so that managing and operating infrastructure must be a central component of this system. There are several steps necessary to understand the problem and be able to choose the most appropriate design. In this chapter we present the information required to build this system:

- Understanding and organizing the information available on the network controllers according to a set of data models;
- Analyse the available protocols for handling the transport between the network controllers and the Graphical User Interface;
- Decide on the GUI server back-end, and how the visual interface should look like.

5.1 Data models

Data models are abstract concepts that map the properties of entities and organizes their attributes, and how they relate to each other. To create a switch management interface, the entities we want to model are the switches themselves, with attributes like the switch name, port counters, and the relationships of the data will allow us to display the links and topology of the network. One of the considerations that were taken into account when choosing the data models was the compatibility with standardized data models by the organizational entities like the IETF [1] and OpenConfig [2].

The *NETCONF* [68] network configuration model, which we explore further in 5.2.1 also defines a data modelling language known as *YANG*, which is used in this protocol to model its configuration and data, and the remote procedure calls [67]. YANG data model defines the hierarchy of data between a NETCONF client and server with the objective of smooth integration with the existing system's infrastructure.

[1] https://www.ietf.org/
[2] http://openconfig.net/

The systems we aim to model are two: the topology between the servers and switches, and the port statistics for each port one the switch. Since there is no data model that would accurately describe both of them correctly, for the topology we chose the IETF network data model [65], and for the port counters the OpenConfig interfaces data model [3].

5.1.1 Topology

The topology data model maps a collection of nodes, and the relationships between each node, called a link. This also allows for describing the network in a vertical hierarchy, by displaying relationships between several layers, which can then be used to display the entire networking stack, for example displaying the physical links between nodes, their connections at layer 2 and layer 3 of the OSI model, and the virtualised relationships that the elements could have in a cloud deployment. As the development of the product continues, and more features are added, for example, layer 3 routing, then we require a flexible data model that can be extended to support the new capabilities.

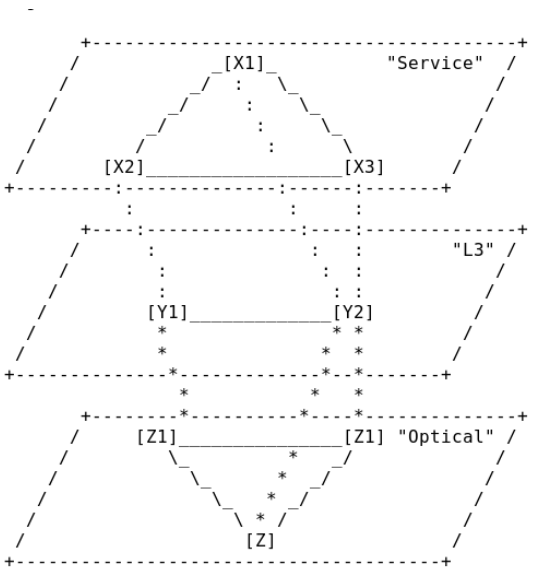


Figure 5.1: Example topology hierarchy achievable with this data model [65]

Mapping the data model to the real world data is then adding the two types of information the data model expects: the first one composed of adding the different networks that composed the entire topology, including their nodes and network types; and then using the previous information to build the links between each of the nodes, using the termination points the model exposes. As seen in 5.1, this data model can be extended by adding underlying networks, representing the several layers in the networking stack.

Displaying the topology proved useful for CAWR since this controller is directly connected to the underlying switches and can see the links among these networking devices. The connection to the servers

[3] http://ops.openconfig.net/branches/master/docs/openconfig-interfaces.html

can also be monitored, by configuring LACP on the servers interface to report their status. Figure 5.2 presents this entire data model, as defined by the IETF.

```
module: ietf-network-topology
augment /nw:networks/nw:network:
   +--rw link* [link-id]
      +--rw link-id            link-id
      +--rw source
      |  +--rw source-node?   -> ../../../nw:node/node-id
      |  +--rw source-tp?     -> ../../../nw:node[nw:node-id=current()/+
      |                       ../source-node]/termination-point/tp-id
      +--rw destination
      |  +--rw dest-node?   -> ../../../nw:node/node-id
      |  +--rw dest-tp?     -> ../../../nw:node[nw:node-id=current()/+
      |                      ../dest-node]/termination-point/tp-id
      +--rw supporting-link* [network-ref link-ref]
         +--rw network-ref    -> ../../../nw:supporting-network/+
         |                    network-ref
         +--rw link-ref       -> /nw:networks/network+
                              [nw:network-id=current()/../network-ref]/+
                              link/link-id
augment /nw:networks/nw:network/nw:node:
   +--rw termination-point* [tp-id]
      +--rw tp-id                           tp-id
      +--rw supporting-termination-point* [network-ref node-ref tp-ref]
         +--rw network-ref    -> ../../../nw:supporting-node/network-ref
         +--rw node-ref       -> ../../../nw:supporting-node/node-ref
         +--rw tp-ref         -> /nw:networks/network[nw:network-id=+
                              current()/../network-ref]/node+
                              [nw:node-id=current()/../node-ref]/+
                              termination-point/tp-id
```

Figure 5.2: The IETF description for the nodes and links in the proposal for network topologies [65]

5.1.2 Port statistics

Modelling the port statistics to build a management interface requires first understanding of the OpenFlow statistics.

The chosen data model should then accurately model the fields that we need to expose, and the data type of counters we wish to measure. In this case, the prevalence of other controllers allows to use the same data models present in their implementations. OpenConfig maintains a set of vendor neutral data models, written in YANG, allowing network operators to use standardized models for their networking infrastructure. The entire set of published models can be accessed in their github page [4].

5.2 Protocols

None of the controllers had a clear way of obtaining the statistics apart from manually looking in the terminal and following the logs exposed and waiting for the appropriate output. In this section we describe the two *Remote Procedure Call (RPC)* systems that were researched, and focus on the advantages which led to the final decision of implementing gRPC on Basebox.

[4] https://github.com/openconfig/public

5.2.1 NETCONF

IETF developed a protocol that allowed for installation, manipulation and deletion of configuration of networking devices called NETCONF, which devices use to expose a full API to their systems. This protocol is set in a client/server communications pattern and is based in the four layers, as can be seen in the image 5.3. Data models and operations, covered in detail in the previous section, are related to the Content layer on the image.

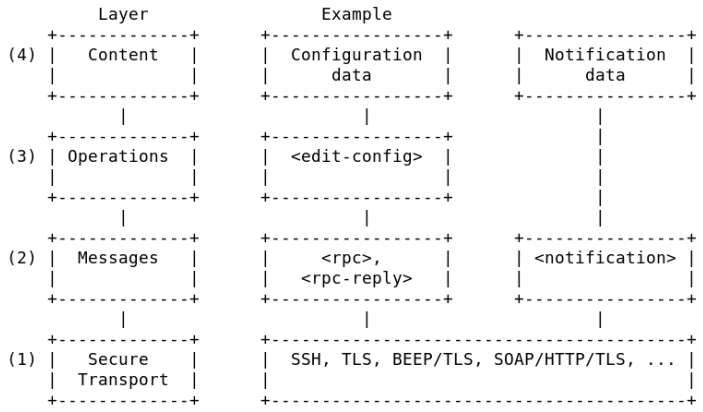


Figure 5.3: NETCONF protocol layers [68]

Configuration of a network device can be complex, and managing separate configurations between device startup and normal operation is a difficult task, but there is occasional need for this capability. NETCONF defines the existence of different *datastores* to enable this feature, allowing the network operator to set an initial configuration, used when the device is initialized, and switching to the running datastore when the device is ready to maintain normal operation. This concept of datastores also enables the creation of a candidate datastores, providing the capability of testing configurations on the network device, checking for any possible errors, while making sure that there is no impact on the current configuration of the device. After the changes have been tested and validated, a <commit> operation can be used to deploy the new configuration to the running datastore.

Another useful feature of the NETCONF protocol, is the possibility of using the rollback-on-error capability. When rolling a new change, and if the system is enabled to support this feature, NETCONF can detect errors in the changes done to the configurations, and return the system to the previous state that is error free.

The NETCONF API provides several operations to interact with the managed devices to get system information and push new configurations. The set of supported operations in the base NETCONF protocol are [68]:

Table 5.1: NETCONF Operations

get
get-config
edit-config
copy-config
delete-config
lock
unlock
close-session
kill-session

NETCONF is able to run on top of several transport protocols. However, NETCONF requires that a persistent connection is maintained between devices, and this connection should be reliable, and support transmission failure. In addition, the security should be handled by the transport layer [66], providing the guarantee that transactions are done in a cryptographically secure channel, between two authenticated hosts. As a result, typical NETCONF implementations are based on SSH or TLS protocols.

5.2.2 gRPC

The basic idea behind RPC systems is defining an interaction between remote environments where subroutines are executed as if they were executed as a local procedure call. These subroutines are defined by specifying the methods that can be called, with their parameters and return values.

gRPC defines a data serialization format known as protocol buffers, or *protobuf*, which is used for defining the service interface and the structure of the messages. This system is based in HTTP/2, due to the optimizations present in this protocol, like

- header compression, which reduces overhead in transmission;
- multiplexing of multiple requests over single connections;
- and stream prioritization, which allows the creation of streaming RPCs, where a bi-directional sequence of messages can be exchanged.

There are some projects [5] that enable the translation between YANG to protobuf, which allows us to use the data models previously chosen, only adding the extra step to convert the files.

Data serialization is a common task for communication between services, and optimization of this task, specifically regarding the speed, allows for reducing overheads in the transmission of the data. A comparative study regarding several serialization formats reports [69], without any optimizations, protobuf improves performance on serializing and de-serializing messages compared to XML or JSON.

[5] https://github.com/openconfig/goyang

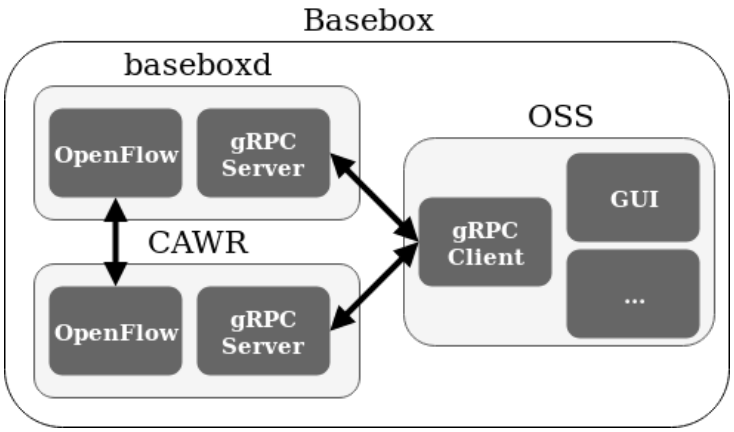


Figure 5.4: Operations Support System architecture

5.2.3 Comparison

Despite of both protocols capability of meeting the requirements that were presented to us, the gRPC framework was chosen due to several reasons:

- Both frameworks allow us to use the standardized data models currently proposed by the IETF and OpenConfig;
- NETCONF trades information as XML encoded information; while gRPC allows to handle information in a way thats native to the language implementation of the client/server;
- The faster serialization of data reduces overheads in the connection and load on the controllers;
- The integration with the existing system was easier: since gRPC has implementations for the languages that the controllers are developed on (i.e. C++), this framework was easier to implement than NETCONF, which would have required integration with third party tools.

5.3 Implementation

By combination of the technologies presented in this chapter, we developed an Operations Support System (OSS), visible in figure 5.4. The controllers had to be extended to support the gRPC interfaces, and this connection is accessible through port 5000 on baseboxd and port 5001 on CAWR.

To demonstrate the interfaces capability of exposing the topology and statistics information, we have developed a simple Graphical User Interface (GUI) as a proof-of-concept of this platform. This GUI can be accessed via a web page, since this provides an ease of access to the system. The web page was developed on top of the Django Framework [6], since it is a framework developed in the Python

[6] https://www.djangoproject.com/

programming language, support for official gRPC integration and ease of implementation. For drawing the topology we used the JavaScript library D3.js [7].

5.3.1 Testing

Testing of this component was done on the setup displayed in image 5.5. This test bed was designed for testing the Basebox controllers, with configuration of the bonded interfaces on the servers connected to both switches, represented as BX on the image.

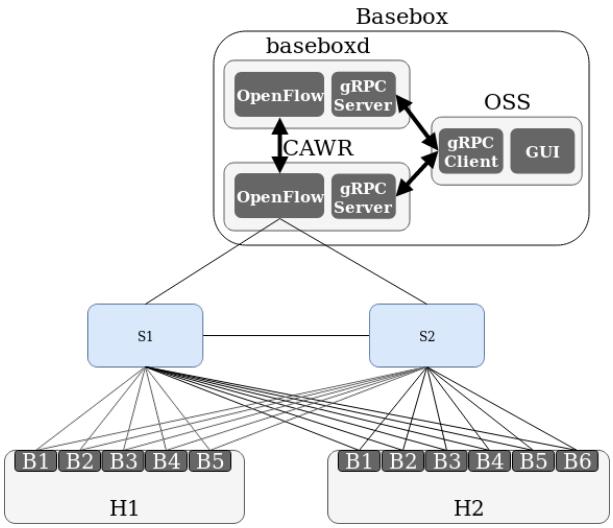


Figure 5.5: Graphical User Interface test setup

[7] https://d3js.org/

5.3.2 Proof-of-concept

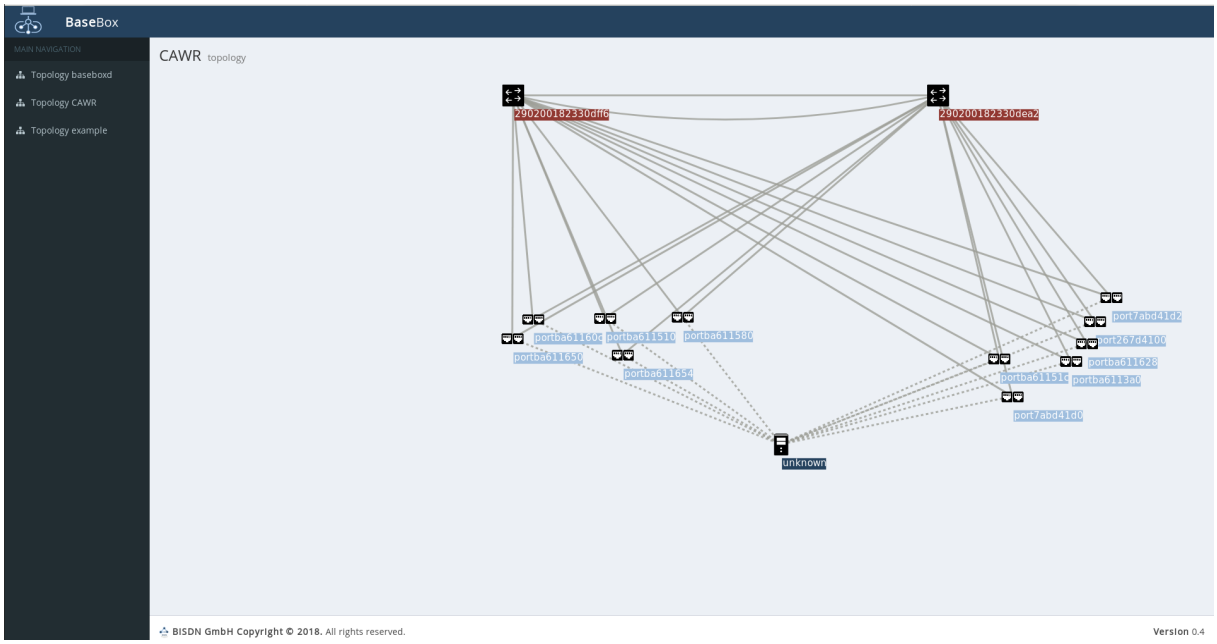


Figure 5.6: Topology obtained from CAWR

Figure 5.6 displays the topology that CAWR reports, similar to the one in the testing environment 5.5. Table A.1 displays the meaning of the icons present in the GUI. Since this system is to be used in a real time deployment, there is no interest in storing the state of the network, which means that the topology must be drawn with every request to the server. Currently the topology is updated every two seconds, providing a balance between real time updates and the interval for requests to the controllers.

A drawback present in figure 5.6 is that the controllers do not have the mapping between each interface and the physical host, which is why the connected server is displayed as unknown. This is a known limitation of the system, that can be solved either by manually updating the host name to port mapping in a configuration file, or extending the controllers to interpret LLDP messages from the servers, although this was considered out-of-scope for this proof-of-concept.

The connection between both controllers to the GUI provides the view for both controllers, which means that CAWR will present the view for the physical switches, bonded ports and hosts, while the baseboxd only shows the giant switch created by CAWR. An advantage of this is the analysis of the global view of the state of the network, and further possibilities include the addition of displaying configured VLANs in each port, and even provide a way to configure these VLANS via a GUI. Interaction with the nodes is possible, and clicking on each provides an insight to the statistics related to the ports in that node, as seen in figure 5.7.

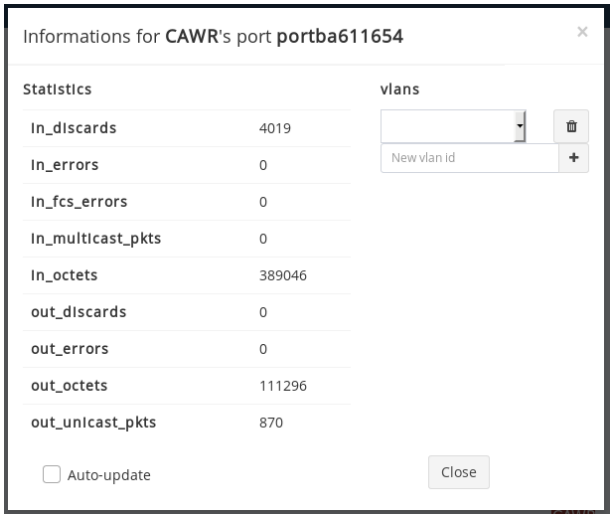


Figure 5.7: Single Port statistics from Baseboxd

Elephant Flow Monitoring

6.1 Elephant flow detection algorithm

Section 3.2.2.4 presents several techniques for time series analysis and change detection, in which we explore the available techniques for modelling the data in a time series, calculating predictions from a time series' historical data (see equation 3.3), and finding change points in this data (see section 3.2.2.4). Using this mathematical baseline, we built an algorithm that monitors the changes in the traffic characteristics of an interface of a switch.

Algorithm 1 is a high level overview of the steps required to obtain the detection, and in the following sections, we introduce each step, and provide some clarifications on the design decisions.

Algorithm 1 Elephant Detection Algorithm - High Level

```
1:  procedure ELEPHANT FLOW DETECTION
2:      Initialization
3:      Query controller
4:      loop
5:          Calculate prediction error
6:          Predict next values
7:          Detection
8:          if Detection then
9:              Raise Alarm
10:         end if
11:         wait 2 seconds
12:     end loop
13: end procedure
```

6.1.1 Initialization

In equation 6.1, B_{XX} and P_{XX} describe to the port statistics obtained from the controllers (see table 2.1), the byte (B_{XX}) and packet (P_{XX}) counters, respectively, and the indexes describe if the counters account

for the transmitted or received data in that port.

$$x_i = \begin{bmatrix} B_{RX} \\ P_{RX} \\ B_{TX} \\ P_{TX} \end{bmatrix} \tag{6.1}$$

The initialization step of the algorithm is a crucial step for obtaining correct results in the algorithm. Algorithm 2 defines the actual steps in this algorithm, allowing the correct initialization of the model parameters, including the trend component, and to provide a baseline for the expected traffic on the network. It is assumed that no traffic anomalies exist during this stage, but a longer period for initialization can account for short bursts of higher traffic.

Algorithm 2 Elephant Detection Algorithm - Initialization

```
1: procedure INITIALIZATION
2:     initialization period = 30s
3:     while t <= initialization period do
4:         x = Query controller
5:         initialization measures += x
6:     end while
7:     Linear Regression (initialization measures)
8:     return Linear regression coefficient
```

6.1.2 Prediction and error calculation

Time series analysis can generate forecasts for future values, assuming the temporal behaviour is maintained for future observations. A change detection mechanism analyses the difference between the predicted value to the observation. In this section we present the prediction and error calculation sections of the elephant flow detection algorithm.

For calculating forecasts, we have presented in section 3.2.2 two possible methods for generating predictions. During the design phase of the algorithm, we selected the exponential smoothing technique, since this is a commonly used technique [29, 30] in the reviewed literature, and provides a generally simple way to generate forecasts based in historical data. For readers convenience, the prediction equations are repeated here from section 3.2.1, since they provide the mathematical baseline for the development of prediction module of the algorithm. The equations are:

$$\begin{aligned} &\hat{x}_1 = x_0, \\ &\hat{x}_t = \alpha x_t + (1-\alpha)\hat{x}_{t-1}, t > 1. \end{aligned} \tag{6.2}$$

With this method are able to predict the values that are expected in the following sample. In this step, the most important consideration is the adjustment of the forgetting factor, $\alpha(0 < \alpha < 1)$, that determines the impact of previous samples on the calculation of the prediction. The value of this factor must be adjusted experimentally, by tuning the value according to the desired output.

6.1.3 Detection

The detection component of the algorithm provides the logic for finding the timestamps where a port's traffic behaviour changes. For this component we consider two possible techniques: the first, which simply compares the output of the error calculation to a certain threshold; and a second, using the CUSUM algorithm (see section 3.2.2.4).

The first analysed detection method is defined as

$$\varepsilon^2 \geq \delta,$$

and this method provides advantages mainly due to the simplicity of the technique, however previous knowledge of the change is required to determine a possible threshold δ.

The second analysed method is the CUSUM algorithm. This method is used for monitoring parameters of a sample, by monitoring deviations of the observations according to a certain target value. Typical implementations of this algorithm are based in an offline approach, calculating the alarm times with knowledge of the entire data set. Since our method is expected to run in an online approach, the adaptation of this algorithm for an online form is based in the application of a sliding window of observations.

The offline method allows us, however, to obtain a baseline for the online detection algorithm. We have collected the error prediction output, and applied the algorithm to a generated data set. Performing this step baselines the expected number of alarms that are raised in a certain testing conditions, so that application of the online algorithm may provide the closest approximation to the optimal offline version. The chosen CUSUM algorithm implementation used was from https://github.com/demotu/BMC, and figure 6.1 shows the alarms raised. Furthermore, this implementation also provides us with the timestamps corresponding to the start and end of the detected changes. This algorithm calculates the CUSUM of a set of data, and expects two inputs: a threshold parameter, which defines the amplitude threshold for the change in data; and the drift parameter tunes the algorithm to allow faster detection (larger drift), or less false alarms (smaller drift).

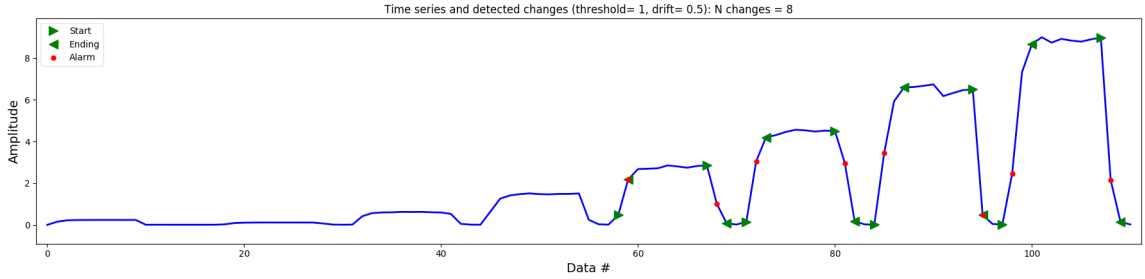


Figure 6.1: Offline CUSUM output

The adaptation of the CUSUM algorithm for utilization as an online technique is based on a sliding window that is updated with every new sample. Applying this method has the advantages of using the CUSUM algorithm without needing extensive changes, while also reducing the amount of memory needed to apply this method. Choosing the window size is a central point to a successful implementation of the change detection mechanism, that is explored further in section 6.3.

6.2 Testing

The design of a testing environment for testing of the change detection algorithm, must allow for the accurate simulation of the traffic conditions on real DCNs, and should provide the flexibility to understand and change the underlying topologies. This indicates a strong motivation for deploying a testing environment in a virtualised environment, using tools like *mininet*, which provides a miniature network that can be changed as needed. This testing suite provides a strong alternative to deploying these changes in hardware.

Despite the changes implemented to Basebox, utilizing these controllers in combination with the virtualised environment poses a challenge, related to the implementation of the OpenFlow protocol in the hardware and software switches. Hardware switches that were used for the implementation of the OSS have a modified version of the OpenFlow tables structure, OF-DPA (see section 2.2.1.1), and the libraries that make up the controller are designed around this. To utilize the controllers, changes to Open vSwitch or other alternatives steps would be required in order to use the environment, which was decided out-of-scope for this thesis. Also, the limited access to the hardware setup meant that an alternative solution was required.

To solve this issue we adopt a different controller for interaction with the virtual environment. In addition, researching the traffic engineering modules in other controllers provides ideas that can later be adopted in Basebox. The chosen controller was Floodlight (see section 2.2.3.2), since it exposes a simple REST API for obtaining statistics, setting table rules, and is continuously maintained providing an optimal test bed.

These elements compose the testing environment that seen in 6.2. To ease the installation of the utilized applications, we based these applications on VMs and containers, and the installation files can be found on github [1].

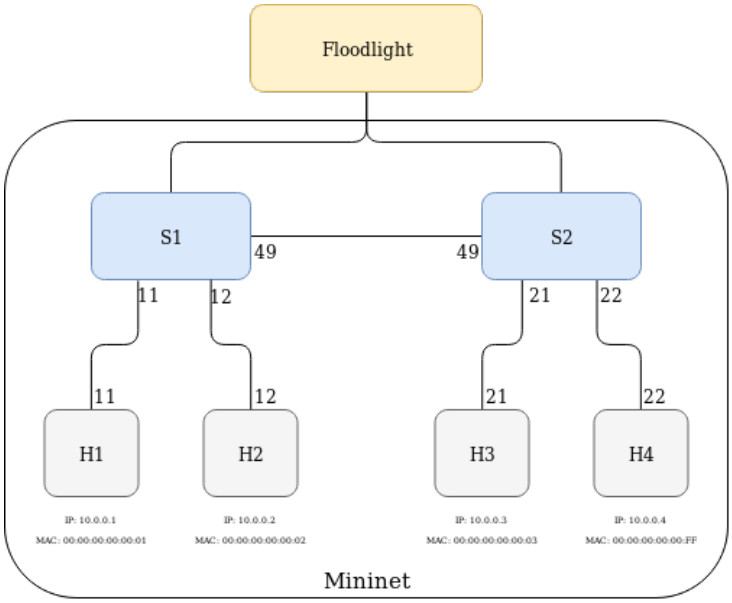


Figure 6.2: The high level overview of the testing setup

Figure 6.2 describes the testing setup designed for testing. This setup provides a close approximation to setups used to the Top-of-Rack switches connected to the servers on the edge layers (see figure 2.2) of data centers, while keeping the resource consumption of the virtualised network and controller to a minimum. In the diagram, the hosts are shown using the H_X notation, ranging from 1 to 4, and the switches use the S_X notation. Information about the hosts, like the IP and MAC addresses can also be seen, and the port numbers used are also displayed.

Mininet provides an API to interact with a virtual network and setup tests in a predictable manner. We utilize this API to develop a script that creates the network topology. For traffic generation, however, we require another tool that provides specific functionalities like control of the inter-packet interval and the packets payload size. The tool *hping* is a common tool used for packet generation and port mapping, and its simplicity allows to quickly deploy different tests against the designed setups. In picture 6.3 the command line interface of the tool is presented and the arguments to build the tests are presented.

[1] https://github.com/rubensfig/thesisdoc.git

```
## hping tests:
## -i uX -> Wait the specified number of seconds or micro seconds between sending each packet
##          the default is to wait one second between each packet, -i uX waits X micro seconds
## -q    -> Quiet output, shows only the result of the test (packet loss, ...)
## -s    -> Sets the source port to a fixed value
## -k    -> Sets the destination port to a fixed value
## -d X  -> Sets packet body size, in addition to the header \
##          if X = 0    : 40 headers + 0 data bytes
##          if X = 500M: 40 headers + 500 data bytes
#timeout 30s hping 10.0.0.2 -i  u$i -q -s 9000 # TEST ONE
##timeout 30s hping 10.0.0.2 -i  u$i -q -s 9000 -k # TEST TWO
##timeout 30s hping 10.0.0.2 -i  u$i -q -s 9000 -d 500M #TEST THREE
```

Figure 6.3: hping tool tests and command line arguments

We consider a test scenario, where every host is communicating with each other, simulating background traffic. This has the purpose of simulating low traffic intensities in the network, and verify the behaviour of the designed algorithm in an approximation to real conditions. In this test scenario, we initialize the network with the background traffic for a specified amount of time, where we assume no abnormal traffic conditions. After this initialization period, we generate traffic between the hosts H_4 and H_2 during a period of 30 seconds, increasing the payload of the packets each time.

The utilization of Open vSwitch has, however some limitations that should be address when the tests are being designed. Figure 6.4 shows the increase of the packet loss as the number of flows in OVS increases. This limitation implies that in order for the tests to run successfully, there needs to be a care in the amount of flows that are generated.

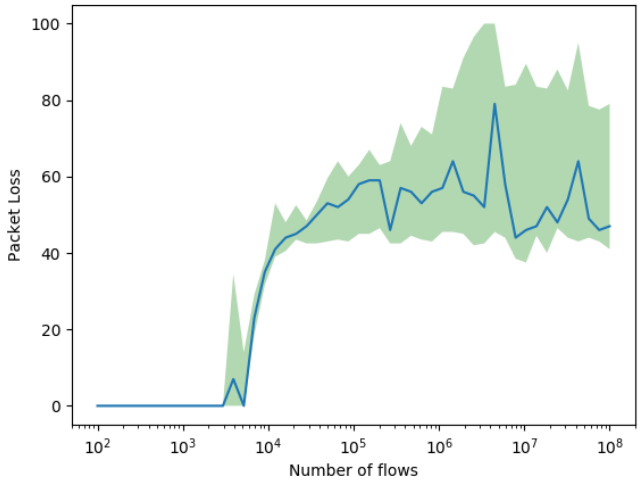


Figure 6.4: OVS measured packet loss

6.3 Results and Evaluation

We analyse the measurements of the initial received bytes in the different ports of the switch. Figures 6.5 and 6.6 provide an insight on the initial traffic characteristics of each port, and we observe that the

ports providing the connection between the two switches report a higher utilization in number of bytes and packets received. This figure also gives an insight on the present trend in the data, that in order to improve the next values' prediction and error calculation, these trends should be removed.

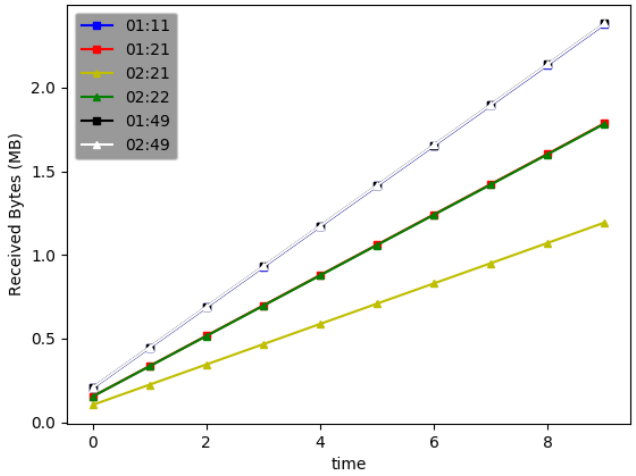


Figure 6.5: Plotting the initial measurements of B_{RX}

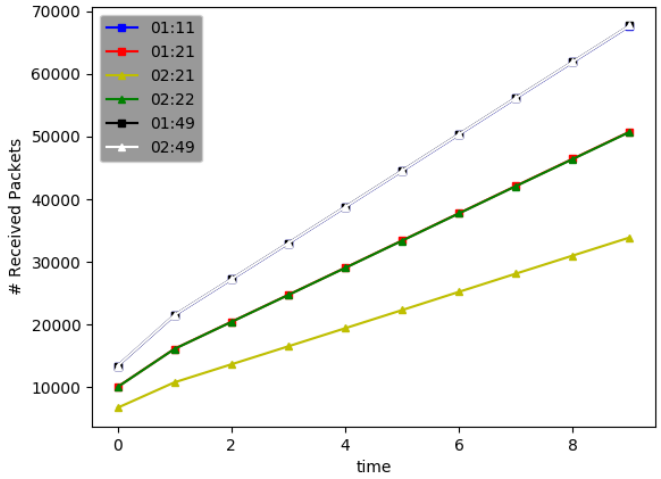


Figure 6.6: Plotting the initial measurements of P_{RX}

For calculating the prediction error ε_t, we analysed two possibilities. The first, obtained with

$$\varepsilon_t = (x_i(t)/\hat{x}_i(t))^2, \tag{6.3}$$

provides the output present in figure 6.7, and while it clearly indicates the deviations created by the elephant flows, it does not provide insight to the level of change caused by larger payloads in the flows.

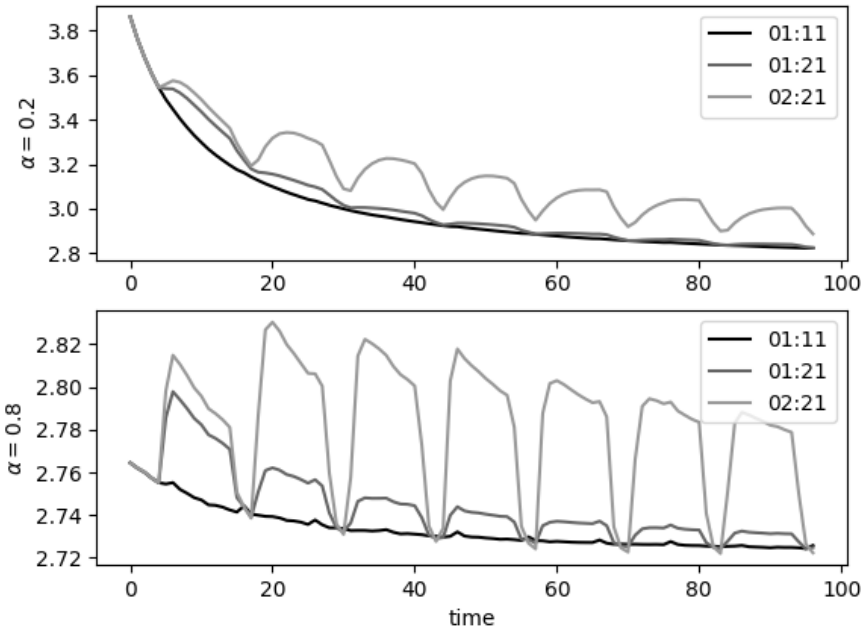


Figure 6.7: Error calculation, with two different values for α, obtained with equation 6.3

The alternate approach devised was to consider the squared prediction errors, as shown in equation 6.4.

$$\varepsilon_t^2 = (x_t - \hat{x}_t)^2 \tag{6.4}$$

This method allows to obtain the prediction error, while increasing the impact of larger changes to the flow data size, and minimizing smaller changes. In figure 6.8 we can visualise the output of this method, which displays the traffic behaviour in the ports of a switch, and the impact of larger payloads of data is visible, as shown by the increasing level of ε_t.

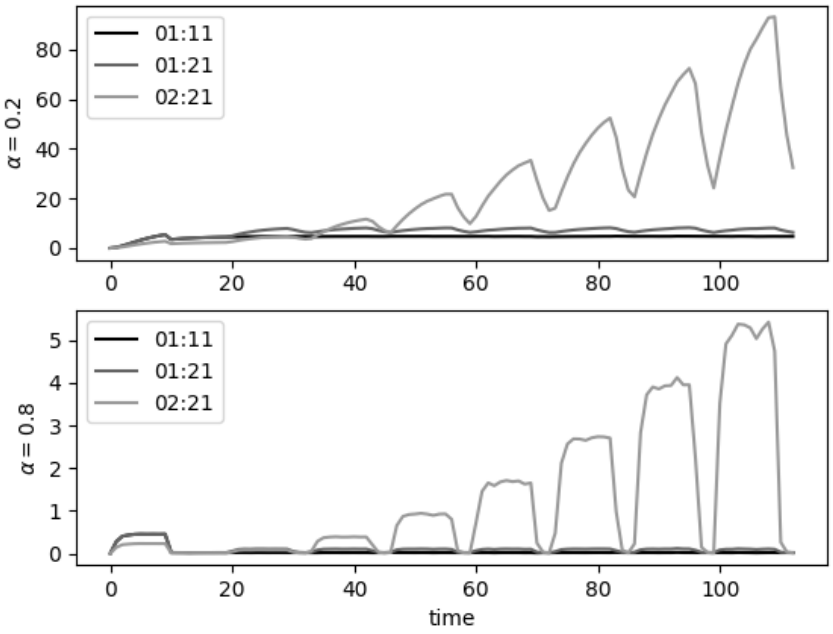


Figure 6.8: ε^2 calculation using the method present in 6.4

In figure 6.7 and 6.8 , we test the error prediction behaviour for two different values of the α parameter of the prediction equations, to see the impact that the previous observations have on the error behaviour. We consider the output of the bottom plot of 6.8 as the desired one, where we assume $\alpha = 0.8$ since it clearly indicates the start and end times of the variation, which in contrast to the upper plot in this figure, where the value $\alpha = 0.2$ is used, does not present a rising trend. Due to this result, we propose higher ($\alpha \to 1$) values for the smoothing factor for posterior results, more specifically $\alpha = 0.8$.

In the initial detection strategy, the error detection is based in comparing the calculated prediction error to a certain threshold. Choosing this threshold requires previous knowledge of the magnitude of the change, and figure 6.9 represents the output of the prediction error calculation using this method. We assume a value of $\delta > 2$ for this detection threshold, to show that the detection threshold may be tuned depending on the use case.

In figure 6.9 we plot the prediction error calculated for every port on the switch, including the inter-switch connection. Since the developed algorithm monitors the changes in traffic characteristics on the switch ports connected to a host, applying the change detection to the inter-switch connection does not provide any useful information, as such, we do not apply the algorithms to these ports.

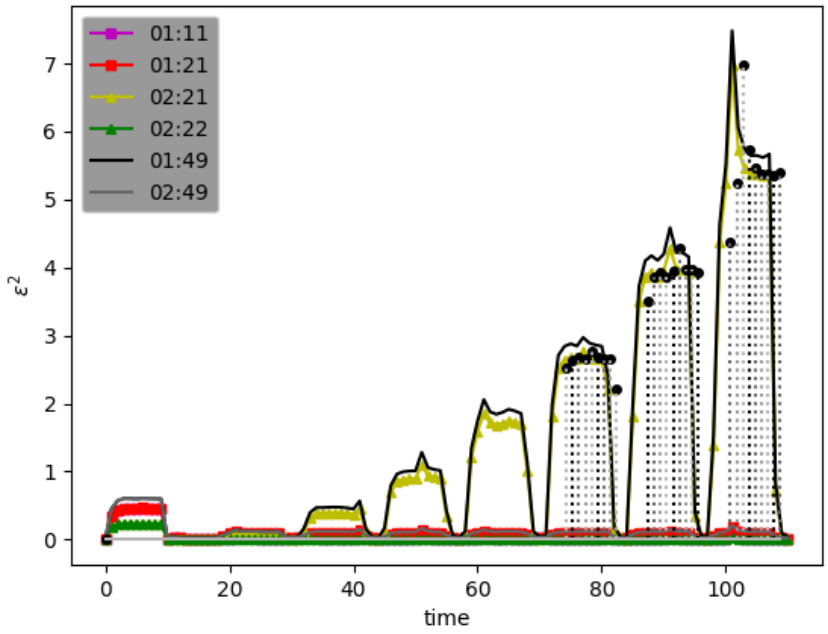


Figure 6.9: Simple detection

Figure 6.10 displays the error detection using the CUSUM algorithm, and the timestamps of alarms raised for the switch port connected to host H_4. From this result, we conclude that with this algorithm we can accurately detect the changes in traffic per port. The greatest difference to the previous mechanism is the reduction in the number of alarms, since this algorithm does the detection across a sliding window of observations, and the alarms are only raised when the traffic characteristics change. Regardless, the same changes in port traffic are detected.

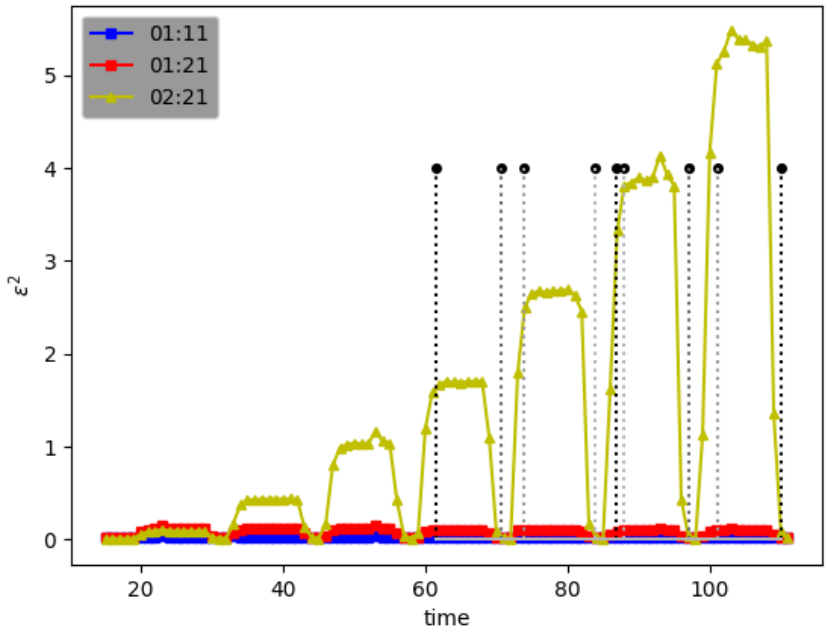


Figure 6.10: CUSUM Detection results

Analysing the output of the algorithm after the tests finish indicates the corresponding timestamps of the alarms. Blindly applying the sliding window technique in the algorithm will result in repeated alarms across the window, and as we vary the length of the window, the algorithm may raise notifications even if the change has taken place at some time in the past. A simple enhancement to the proposed algorithm is to consider only the first occurrence of the alarms. Figure 6.11 shows the result of plotting the number of alarms against the changing size of the window, the upper figure considering only the timestamps of the first occurrence of the alarm, on the contrary of the lower figure.

This proposed approach also allows the comparison between the offline and the online approach to change detection. In figure 6.1, the offline output can be analysed for the number of expected alarms on this testing scenario. Comparing this result to the enhanced version in figure 6.11, we conclude that the number of errors are similar to the offline version, which means that we can accurately detect every change in the test scenario with the online algorithm. In this figure, the shaded area represents the measured variation in the number of alarms.

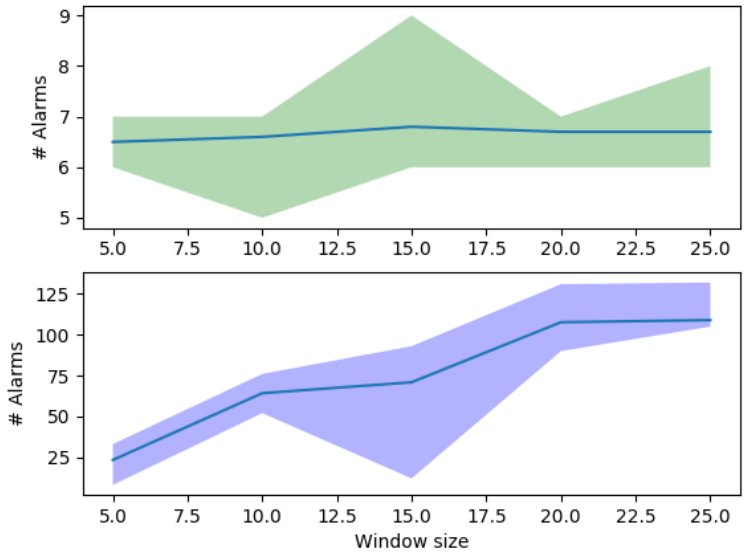


Figure 6.11: Comparison of the alarm count between enhanced and non enhanced version

Continuing the analysis of the impact caused by the variation of the window size, the analysis of a single elephant flow (as shown in figure 6.12) allows us to evaluate the performance of the algorithm, with the methods mentioned in section 3.2.2.5. Obtaining a baseline for the amount of raised alarms and detection timestamps was done by applying the offline CUSUM algorithm to this test case.

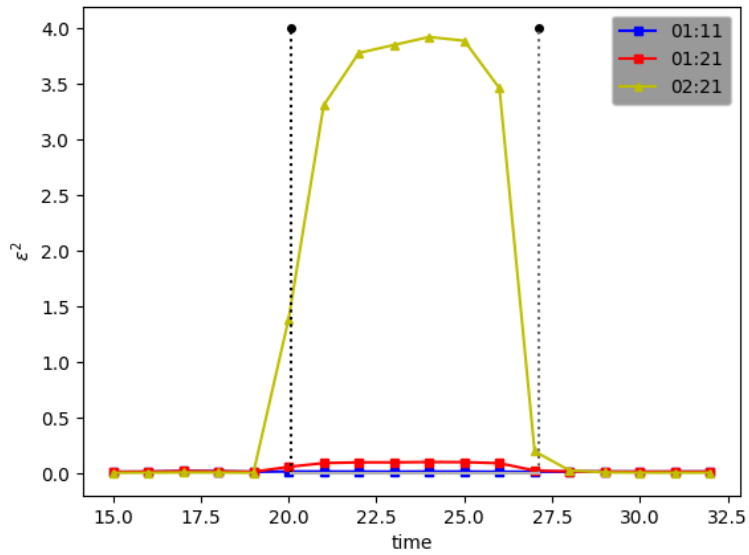


Figure 6.12: Single elephant flow

The analysis of the error detection time will provide us with the performance of this change detection algorithm. By establishing the baseline detection times obtained from the CUSUM algorithm, using the same threshold and drift parameters in the online and offline case, we can determine the difference between these baseline values and the values obtained online. Figure 6.13 shows the result of this analysis, and the variation caused to the time to detection from the varying window size. To generate this result, we have performed 30 different measures for each window size, and compare the time of detection to the baseline value.

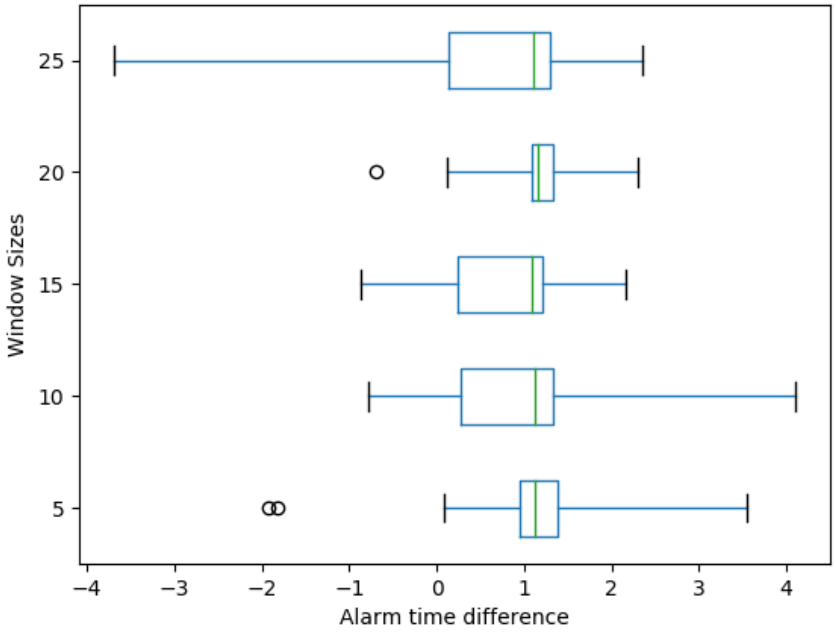


Figure 6.13: Analysis of the time to detect the change

Analysing the single elephant flow, and the output of the online detection algorithm also allows us to compare the baseline amount of alarms. This will give us an insight to the amount of false alarms generated, and the relationship of these with the window size. Table 6.1 shows us the obtained statistics of the number of alarms, which confirms a worse accuracy for the number of alarms with the smaller windows.

Table 6.1: False alarms statistics

Window Size	5	10	15	20	25
count	30	30	30	30	30
$\bar{x}$	2.833333	2.166667	2.0	2.0	2.0
σ^2	0.791478	0.379049	0.0	0.0	0.0

In spite of the expected result of quicker detection for smaller windows [73], figure 6.13 does not present much variation in the detection time, but the amount of false alarms raised by the smaller windows suggest a better performance of the algorithm using larger window sizes.

Finally, an interesting consequence of applying a change detection method applied to the statistics on the ports on the switch, is the independence of the protocol, allowing us to understand traffic changes on the ports regardless if the transport protocol is TCP or UDP.

Conclusion

7.1 Summary of Results

The main objective for this thesis was building a management system that would integrate with a pre-existing Software-Defined Network controller, exposing information for network operators to manage and configure their networking infrastructure. With these requirements, we have proposed a management environment that extends a previously existing system, by adding an interface to baseboxd and CAWR, two SDN controllers composing the Basebox environment, allowing further developments in the field of Traffic Engineering with these systems. Integrating this system, we have designed a Graphical User Interface for interaction with the users, allowing for simple visualisation of the network's physical topology, and the display of interfaces' statistics, like the packets and bytes received and sent, or the number of errors.

We have also proposed an algorithm that allows for monitoring traffic changes in ports, in order to detect elephant flows in the network. Despite not having used the Basebox system for testing this algorithm, due to differences in the testing environment, we believe that the same algorithm can be used for large flow detection in the Basebox stack, by changing the interface for obtaining statistics. We have shown that a simple method can be employed by operators to monitor the state of their network, and rely on this algorithm to provide them with alarms of port changes.

7.2 Future Work

Despite our conclusion that the main objectives of the thesis were achieved, the large scope of themes that this topic encompasses means that not everything could be successfully covered. As for more immediate concerns, the next steps in guaranteeing a stable product would be the expansion of the GUI to report and configure VLANs in each port that is monitored; and support layer 3 functionality, such as visualising next hop neighbours, routing tables, etc. In regards to longer term goals, continuing the work on monitoring not only the port change, but the actual flow that contributes to the largest changes in ports. This could be expanded into a system that analyses the services and applications that contribute the most to the traffic volumes in the network, which can then be further optimized by reporting the periodicity

of the largest traffic volumes. The aspect of providing a system that can discriminate the traffic by transport protocol is also an interesting research topic for improving Quality of Service in Software-Defined Networks.

Appendix

A.1 GUI icons mapping

Table A.1: GUI icons and their meaning

	switch
	host
	port

Bibliography

[1] David Sims. Carousel's Expert Walks Through Major Benefits of Virtualization, July 2011.

[2] Altino Manuel Silva Sampaio. *Energy-efficient and SLA-based Management of IaaS Cloud Data Centers*. PhD Thesis, Universidade do Porto (Portugal), 2015.

[3] Taimur Bakhshi. *User-Centric Traffic Engineering in Software Defined Networks*. PhD Thesis, University of Plymouth, 2017.

[4] Open Networking Foundation. SDN Architecture Overview. Technical TR_SDN ARCH Overview 1.1 1111 2014, Open Networking Foundation, November 2014.

[5] Myung-Ki Shin, Ki-Hyuk Nam, and Hyoung-Jun Kim. Software-defined networking (SDN): A reference architecture and open APIs. pages 360–361. IEEE, October 2012.

[6] Open Networking Foundation. OpenFlow Switch Specification Version 1.3.5 (Protocol version 0x04), March 2015.

[7] Broadcom Corporation. OpenFlow^{TM} Data Plane Abstraction (OF-DPA): Abstract Switch Specification, January 2017.

[8] Sakir Sezer, Sandra Scott-Hayward, Pushpinder Kaur Chouhan, Barbara Fraser, David Lake, Jim Finnegan, Niel Viljoen, Marc Miller, and Navneet Rao. Are we ready for SDN? Implementation challenges for software-defined networks. *IEEE Communications Magazine*, 51(7):36–43, 2013.

[9] Danny Raz, International Federation for Information Processing, Institute of Electrical and Electronics Engineers, Communications Society, and Computer Society, editors. *2014 10th International Conference on Network and Service Management (CNSM 2014): Rio de Janeiro, Brazil, 17 - 21 November 2014 ; [including workshop papers]*. IEEE, Piscataway, NJ, 2014. OCLC: 931888885.

[10] Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen, Katia Obraczka, and Thierry Turletti. A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Communications Surveys & Tutorials*, 16(3):1617–1634, 2014.

[11] Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, and Suparna Bhattacharya. Effective switch memory management in OpenFlow networks. pages 177–188. ACM Press, 2014.

[12] Kevin Phemius, Mathieu Bouet, and Jeremie Leguay. DISCO: Distributed multi-domain SDN controllers. pages 1–4. IEEE, May 2014.

[13] Rahamatullah Khondoker, Adel Zaalouk, Ronald Marx, and Kpatcha Bayarou. Feature-based comparison and selection of Software Defined Networking (SDN) controllers. pages 1–7. IEEE, January 2014.

[14] Pankaj Berde, William Snow, Guru Parulkar, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, and Pavlin Radoslavov. ONOS: towards an open, distributed SDN OS. pages 1–6. ACM Press, 2014.

[15] Project Floodlight. Floodlight Is an Open SDN Controller, 2017.

[16] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. OpenDaylight: Towards a Model-Driven SDN Controller architecture. pages 1–6. IEEE, June 2014.

[17] Andy Bierman, Martin Bjorklund, and Kent Watsen. *RESTCONF Protocol*. Number 8040 in Request for Comments. RFC Editor, January 2017. Published: RFC 8040.

[18] Ian F. Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. Research challenges for traffic engineering in software defined networks. *IEEE Network*, 30(3):52–58, May 2016.

[19] Seyed Mohammad Mousavi and Marc St-Hilaire. Early detection of DDoS attacks against SDN controllers. pages 77–81. IEEE, February 2015.

[20] Andrew R. Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-overhead datacenter traffic management using end-host-based elephant detection. In *INFOCOM, 2011 Proceedings IEEE*, pages 1629–1637. IEEE, 2011.

[21] Daniela Brauckhoff, Bernhard Tellenbach, Arno Wagner, Martin May, and Anukool Lakhina. Impact of packet sampling on anomaly detection metrics. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 159–164. ACM, 2006.

[22] Recommendation X.700 MANAGEMENT FRAMEWORK FOR OPEN SYSTEMS INTERCONNECTION (OSI) FOR CCITT APPLICATIONS, September 1992.

[23] Mark Fedor, James R. Davin, Martin Lee Schoffstall, and Dr Jeff D. Case. *Simple Network Management Protocol (SNMP)*. Number 1157 in Request for Comments. RFC Editor, May 1990. Published: RFC 1157.

[24] Dr Marshall T. Rose and Keith McCloghrie. *Structure and identification of management information for TCP/IP-based internets*. Number 1155 in Request for Comments. RFC Editor, May 1990. Published: RFC 1155.

[25] Dr Marshall T. Rose and Keith McCloghrie. *Management Information Base for network management of TCP/IP-based internets*. Number 1156 in Request for Comments. RFC Editor, May 1990. Published: RFC 1156.

[26] Jürgen Schönwälder. *Overview of the 2002 IAB Network Management Workshop*. Number 3535 in Request for Comments. RFC Editor, May 2003. Published: RFC 3535.

[27] Lucian Popa, Sylvia Ratnasamy, Gianluca Iannaccone, Arvind Krishnamurthy, and Ion Stoica. A Cost Comparison of Datacenter Network Architectures. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 16:1–16:12, New York, NY, USA, 2010. ACM.

[28] BISDN GmbH. Software - Defined Cloud Networking, 2017.

[29] Roman Jašek, Anna Szmit, and Maciej Szmit. Usage of Modern Exponential-Smoothing Models in Network Traffic Modelling. In Ivan Zelinka, Guanrong Chen, Otto E. Rössler, Vaclav Snasel, and Ajith Abraham, editors, *Nostradamus 2013: Prediction, Modeling and Analysis of Complex Systems*, pages 435–444, Heidelberg, 2013. Springer International Publishing.

[30] Gerhard Münz. *Traffic anomaly detection and cause identification using flow-level measurements*. Number 2010,06 in Network architectures and services. Network Architectures and Services, Techn. Univ. München, München, 2010. OCLC: 845870015.

[31] Vivek Shrivastava, Petros Zerfos, Kang-Won Lee, Hani Jamjoom, Yew-Huey Liu, and Suman Banerjee. Application-aware virtual machine migration in data centers. In *INFOCOM, 2011 Proceedings IEEE*, pages 66–70. IEEE, 2011.

[32] Panagiotis Georgopoulos, Matthew Broadbent, Bernhard Plattner, and Nicholas Race. Cache as a service: Leveraging sdn to efficiently and transparently support video-on-demand on the last mile. In *Computer Communication and Networks (ICCCN), 2014 23rd International Conference on*, pages 1–9. IEEE, 2014.

[33] Mohammad Ali Maddah-Ali and Urs Niesen. Fundamental limits of caching. *IEEE Transactions on Information Theory*, 60(5):2856–2867, 2014.

[34] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The cost of a cloud: research problems in data center networks. *ACM SIGCOMM computer communication review*, 39(1):68–73, 2008.

[35] Jing Xu and Jose A. B. Fortes. Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments. pages 179–188. IEEE, December 2010.

[36] Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. Netshare and stochastic netshare: predictable bandwidth allocation for data centers. *ACM SIGCOMM Computer Communication Review*, 42(3):5–11, 2012.

[37] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[38] Arvind Krishnamurthy and B. Plattner, editors. *Passive and active measurement: 11th international conference, PAM 2010, Zurich, Switzerland, April 7-9, 2010: proceedings*. Number 6032 in Lecture notes in computer science. Springer, Berlin, 2010.

[39] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: managing caches in multi-tenant data centers. pages 174–181. ACM Press, 2015.

[40] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-storage virtualization: integration and load balancing in data centers. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 53. IEEE Press, 2008.

[41] Ashwin Lall, Vyas Sekar, Mitsunori Ogihara, Jun Xu, and Hui Zhang. Data streaming algorithms for estimating entropy of network traffic. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 145–156. ACM, 2006.

[42] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9. IEEE, 2010.

[43] Jae-Hyun Jun, Cheol-Woong Ahn, and Sung-Ho Kim. DDoS attack detection by using packet sampling and flow features. pages 711–712. ACM Press, 2014.

[44] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. Flowsense: Monitoring network utilization with zero measurement cost. In *International Conference on Passive and Active Network Measurement*, pages 31–41. Springer, 2013.

[45] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Nsdi*, volume 10, pages 19–19, 2010.

[46] Tatsuya Mori, Masato Uchida, Ryoichi Kawahara, Jianping Pan, and Shigeki Goto. Identifying elephant flows through periodically sampled packets. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 115–120. ACM, 2004.

[47] Jordi Ros-Giralt, Alan Commike, Sourav Maji, and Malathi Veeraraghavan. A Mathematical Framework for the Detection of Elephant Flows. *arXiv preprint arXiv:1701.01683*, 2017.

[48] Anukool Lakhina, Mark Crovella, and Christiphe Diot. Characterization of network-wide anomalies in traffic flows. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, pages 201–206. ACM, 2004.

[49] Dean H. Lorenz, Ariel Orda, and Danny Raz. Optimal partition of QoS requirements for many-to-many connections. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1670–1679. IEEE, 2003.

[50] networkheresy . Of Mice and Elephants, November 2013.

[51] Justin Pettit. Open vSwitch and the Intelligent Edge, 2014.

[52] Huikang Zhu, Hongbo Fan, Xuan Luo, and Yaohui Jin. Intelligent timeout master: Dynamic timeout for SDN-based data centers. In *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*, pages 734–737. IEEE, 2015.

[53] Peng Wang, Hong Xu, Zhixiong Niu, Dongsu Han, and Yongqiang Xiong. Expeditus: Congestion-Aware Load Balancing in Clos Data Center Networks. *IEEE/ACM Transactions on Networking*, 25(5):3175–3188, October 2017.

[54] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[55] Radhika Mittal, David Zats, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, and David Wetherall. TIMELY: RTT-based Congestion Control for the Datacenter. pages 537–550. ACM Press, 2015.

[56] Dean H. Lorenz and Ariel Orda. Optimal partition of QoS requirements on unicast paths and multicast trees. *ieee/acm Transactions on Networking*, 10(1):102–114, 2002.

[57] Md Faizul Bari, Raouf Boutaba, Rafael Esteves, Lisandro Zambenedetti Granville, Maxim Podlesny, Md Golam Rabbani, Qi Zhang, and Mohamed Faten Zhani. Data center network virtualization: A survey. *IEEE Communications Surveys & Tutorials*, 15(2):909–928, 2013.

[58] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. Inside the Social Network's (Datacenter) Network. pages 123–137. ACM Press, 2015.

[59] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Understanding data center traffic characteristics. page 65. ACM Press, 2009.

[60] Ejaz Ahmed, Andrew Clark, and George Mohay. A Novel Sliding Window Based Change Detection Algorithm for Asymmetric Traffic. pages 168–175. IEEE, October 2008.

[61] George E. P. Box, Gwilym M. Jenkins, Gregory C. Reinsel, and Greta M. Ljung. *Time series analysis: forecasting and control*. Wiley series in probability and statistics. John Wiley & Sons, Inc, Hoboken, New Jersey, fifth edition edition, 2016.

[62] Prajakta S. Kalekar. Time series forecasting using holt-winters exponential smoothing. *Kanwal Rekhi School of Information Technology*, 4329008:1–13, 2004.

[63] Anton Beloglazov and Rajkumar Buyya. Optimal online deterministic algorithms and adaptive heuristics for energy and performance efficient dynamic consolidation of virtual machines in Cloud data centers: ENERGY AND PERFORMANCE EFFICIENT DYNAMIC CONSOLIDATION OF VIRTUAL MACHINES. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, September 2012.

[64] Julius Schulz-Zander, Carlos Mayer, Bogdan Ciobotaru, Stefan Schmid, and Anja Feldmann. OpenSDWN: Programmatic Control over Home and Enterprise WiFi. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 16:1–16:12, New York, NY, USA, 2015. ACM.

[65] Alexander Clemm, Jan Medved, Robert Varga, Nitin Bahadur, Hariharan Ananthakrishnan, and Xufeng Liu. A Data Model for Network Topologies. Internet-Draft draft-ietf-i2rs-yang-network-topo-20, Internet Engineering Task Force, December 2017.

[66] Jürgen Schönwälder. Network Configuration Management with NETCONF and YANG, July 2012.

[67] Martin Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. Number 6020 in Request for Comments. RFC Editor, October 2010. Published: RFC 6020.

[68] Rob Enns, Martin Bjorklund, Andy Bierman, and Jürgen Schönwälder. *Network Configuration Protocol (NETCONF)*. Number 6241 in Request for Comments. RFC Editor, June 2011. Published: RFC 6241.

[69] Maxim Novak. Serialization Performance comparison (C#/.NET) – Formats & Frameworks (XML–DataContractSerializer & XmlSerializer, BinaryFormatter, JSON– Newtonsoft & ServiceStack.Text, Protobuf, MsgPack), March 2014.

[70] Bruno Krebs. Beating JSON performance with Protobuf.

[71] Wan Kamarul Ariffin Wan Ahmad and Sabri Ahmad. Arima model and exponential smoothing method: A comparison. *AIP Conference Proceedings*, 1522(1):1312–1321, 2013.

[72] Rob J. Hyndman and Anne B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4):679 – 688, 2006.

[73] Dhruv Choudhary, Arun Kejariwal, and Francois Orsini. On the Runtime-Efficacy Trade-off of Anomaly Detection Techniques for Real-Time Streaming Data. *arXiv preprint arXiv:1710.04735*, 2017.